(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
24 October 2002 (24.10.2002)

PCT

(10) International Publication Number
WO 02/084947 A2

(51) International Patent Classification⁷: H04L 12/14

(21) International Application Number: PCT/US02/06074

(22) International Filing Date: 26 February 2002 (26.02.2002)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
60/271,661 26 February 2001 (26.02.2001) US

(71) Applicant (*for all designated States except US*): 4THPASS INC. [US/US]; Suite 100, 83 South King Street, Seattle, WA 98104 (US).

(72) Inventors; and
(75) Inventors/Applicants (*for US only*): **MEHTA, Samir, Narendra** [IN/US]; Apt. C304, 300 Vuemont Place NE, Renton, WA 98056 (US). **RAMADAN, Mazin** [US/US]; Apt. 313, 303 East Pike Street, Seattle, WA 98122 (US). **NGUYEN, Ngochan, T.** [VN/US]; Apt. N171, 4601 South Graham Street, Seattle, WA 98118 (US). **JANSEN, Markus, L.** [DE/US]; Apt. N203 300 Vermont Place NE, Renton, WA 98056 (US).

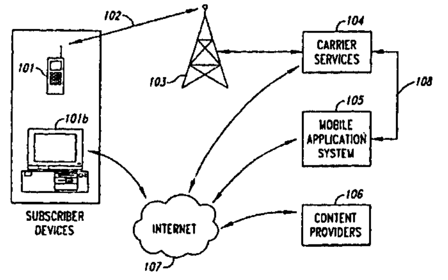
(74) Agent: **BIERMAN, Ellen, M.**; Seed Intellectual Property Law Group PLLC, Suite 6300, 701 Fifth Avenue, Seattle, WA 98104-7092 (US).

(81) Designated States (*national*): AE, AG, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, BZ, CA, CH, CN, CO, CR, CU, CZ, DE, DK, DM, DZ, EC, EE, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD, MG, MK, MN, MW, MX, MZ, NO, NZ, OM, PH, PL, PT, RO, RU, SD, SE, SG, SI, SK, SL, TJ, TM, TN, TR, TT, TZ, UA, UG, US, UZ, VN, YU, ZA, ZM, ZW.

(84) Designated States (*regional*): ARIPO patent (GH, GM, KE, LS, MW, MZ, SD, SL, SZ, TZ, UG, ZM, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE, TR), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, ML, MR, NE, SN, TD, TG).

*[Continued on next page]*

(54) Title: METHOD AND SYSTEM FOR TRANSMISSION-BASED BILLING OF APPLICATIONS



(57) Abstract: Computer- and network-based methods and systems for transmission-based billing are provided. Example embodiments provide a Packet-Based Billing System ("PBBS"), which enables application providers, such as carriers and content providers, to bill subscribers for the use of content on mobile subscriber devices, such as wireless devices, on a per-application, per-user basis based upon the extent of the usage. Embodiments of the present invention can also be used to bill subscribers for the use of content on a per-application, per-user basis for wired subcriber devices as well, using the same techniques. In operaiton, the PBBS provides modified content by inserting billing and tracking code into content returned to a requesting device. The modified content, when executed, tracks the amount of data sent and received between the content and a network and posts the accumulated data to a proxy/billing server according to business rules for an interval/frequency to post such data. The proxy/billing server stores the raw billing data and an accounting program retrieves the billing data to generate customer (call) data records. Business rules that specific different charges for different content or users can be incorporated into the system.

WO 02/084947 A2

**Published:**
— *without international search report and to be republished upon receipt of that report*

# METHOD AND SYSTEM FOR TRANSMISSION-BASED BILLING OF APPLICATIONS

## BACKGROUND OF THE INVENTION

### Field of the Invention

The present invention relates to a method and system for transmission-based billing and, in particular, to methods and systems for billing the use of wireless and wired applications based upon data transmitted over a network.

### Background Information

Today, wireless devices have become prolific in many communities of the world. Devices such as wireless phones, handsets, personal information managers, electronic organizers, personal digital assistants, portable e-mail machines, game machines, and other devices are used by subscribers of telephone carriers to add convenience to our lives. However, the software used on such devices and the mechanisms for deploying and billing for the use such software to these devices are arcane. Typically, applications or other services are billing when they are loaded onto a wireless device (a one-time, flat fee charge); on a subscription basis, such as a charge for the overall use of a set of applications and services; or for total airtime. In newer, third generation, wireless networks, such as GPRs, carriers are able to identify a total number of physical packets or the amount of data used by a device. Thus, billing models for a typical network carrier reflect flat fee or subscription-based billing and do not necessarily accurately reflect use of applications and services. For example, wireless applications that access a network while running on the device, for example a calendar, a browser, or an email client, typically utilize different carrier resources than applications that do not access a network, for example, a calculator or a text editor. A carrier or content provider is unable to accurately charge for the amount of carrier resources a particular application consumes because current billing systems provided at the wireless carrier level are unable to provide differentiated billing on an application level, except at the time of download.

## BRIEF SUMMARY OF THE INVENTION

Embodiments of the present invention provide computer- and network-based methods and systems for transmission-based billing, which provides the tracking

of billing information based upon the volume of data transmitted between applications, or other types of content, and a network. Example embodiments provide a Packet-Based Billing System ("PBBS"), which enables application providers, such as carriers and third party content providers, to bill clients or subscribers (generically "users") for the use of applications/services (generically, "content") on mobile subscriber devices, such as wireless devices, on a per-application, per-user basis based upon the extent of the usage. Embodiments of the present invention can also be used to bill subscribers on a per-application, per-user basis for wired subscriber devices as well, using the same techniques.

The PBBS determines and provides billing tracking code and associated communication support, generally referred to collectively herein as "billing tracking code," to enable application and service providers or other content providers (typically carriers) to automatically track billing information on a configurable, logical packet level, based upon the amount of data sent and/or received by the application or service over the network. In the case of wireless devices and a wireless network, this billing data tracks usage of the network on a per-application basis instead of traditional "total airtime" use models. The data tracked is associated with a particular user, thus allowing the application or service provider to bill on a per-user, per-application basis based upon actual volume of use. In one embodiment, the billing data tracked and posted includes an amount of data sent/received across the network, a timestamp, an application identifier, a security key, a transaction identifier, and a retry expiration indicator. In other embodiments, the billing data may include a subset of these data or may include different or additional data. The application/service provider can subsequently implement a variety of billing policies at the application or user level, which can change over time.

In one embodiment, the PBBS instruments packet-based billing tracking code into the content to track and accumulate billing data on the client device and to post the data to a server, such as a proxy server or a billing server. For some content, such as Java applications, .NET applications, and other binary applications, instrumentation is accomplished at the byte-code level. In one such embodiment, the instrumentation is performed by a content (code) modifier that analyzes the content to determine data structures, calling sequence, and location and identity of any network calls and replaces these calls by proxy network calls that contain the billing and tracking code.

In another embodiment, the billing tracking code is incorporated into the content by modifying the content according to a written specification. In yet another embodiment, the billing tracking code is incorporated into the content through calls to an Application Programming Interface library. In yet another embodiment, the billing tracking code is placed in the network driver software on the client device and interfaces directly to a proxy/billing server.

In yet another embodiment, a security key is inserted into the content to enable the billing tracking code to identify itself to a proxy server when billing data is posted from a client device. In one such embodiment, the security key is a number uniquely associated with each content/user combination and is stored in a secure data repository to prevent misappropriation and false billing data. In another embodiment, the security key is a unique random number.

In one embodiment, the PBBS comprises a content (code) modifier; one or more data repositories containing associations of network calls to different proxy network calls containing billing tracking code, business rules for billing data, raw billing data, and security keys; a proxy server, a billing server, and an accounting program. In accordance with this embodiment, the PBBS functions may be integrated into and dispersed over different components in an application provisioning system. These components then interact to determine and insert billing tracking code into the content; receive data from the billing tracking code; and process the tracked data in conjunction with indicated billing policies to generate billing records. In one embodiment, the billing tracking code is inserted in response to a request from a client device for an application and a modified application is automatically returned. The billing data is then posted when the modified application is executed on the client device. In another embodiment, the billing data is generated directly in response to a request from a client device for streaming content, such as streaming audio and video, since the requests are for a particular number of bytes or amount of data. In one of these embodiments, the application provisioning system provisions applications for wireless devices. In another embodiment, the application provisioning system provisions applications for wired devices.

In another embodiment, the transmission-based data is used to route network packets from applications to other servers (network traffic). According to this embodiment, the transmission-based (billing) data is used to determine the extent of resources an application is consuming so that a proxy server can direct network traffic to promote greater efficiency or, for example, to provide/guarantee better response time for heavily used or popular applications, or applications based upon some other criteria.

## BRIEF DESCRIPTION OF THE DRAWINGS

Figure 1 is an overview block diagram of an example transmission-based billing system.

Figure 2 is an example block diagram of content instrumented with billing tracking code as performed by a content (code) modifier of a Packet-Based Billing System.

Figure 3 is a block diagram of an example transmission-based billing system implemented within an example Mobile Application System.

Figure 4 is an example block diagram of a general-purpose computer system and a client device for practicing embodiments of the transmission-based billing system.

Figure 5 is a block diagram of an example procedure for submitting an application to a Mobile Application System for use with transmission-based billing.

Figure 6 is an example block diagram that illustrates an overall process for transparently delivering an application that supports transmission-based billing to a client device using a Mobile Application System..

Figure 7 is an example flow diagram of a routine for modifying an application to support transmission-based billing.

Figure 8 is an example flow diagram of a routine for analyzing and instrumenting an application with support for transmission-based billing.

Figure 9 is an example block diagram that illustrates an overall process for communicating billing data based upon network transmission data on a per application basis.

Figure 10 is an example flow diagram of a client routine for sending data to a network that has been modified to collect and transmit billing data.

Figure 11 is an example flow diagram of a client routine for receiving data from a network that has been modified to collect and transmit billing data.

Figure 12 is an example flow diagram of a server routine for processing posted transmission-based billing data.

Figure 13 is an example flow diagram of steps for generating billing charges based upon transmission-based billing data.

## DETAILED DESCRIPTION OF THE INVENTION

Embodiments of the present invention provide computer- and network-based methods and systems for transmission-based billing, which provides the tracking of billing information based upon the volume of data transmitted between applications, or other types of content, and a network. Example embodiments provide a Packet-Based Billing System ("PBBS"), which enables application providers, such as carriers and third party content providers, to bill clients or subscribers (generically "users") for the use of applications/services (generically "content") on mobile subscriber devices, such as wireless devices, on a per-application, per-user basis based upon the extent of the usage. Embodiments of the present invention can also be used to bill subscribers on a per-application, per-user basis for wired subscriber devices as well, using the same techniques. Although this description primarily refers to applications, one skilled in the art will recognize that the methods and systems described herein are applicable to any other type of content that can be transmitted at a packet level across a network, such as services and resources, and that is capable of communicating billing data to a server when it is "executed" on a client device. For example, an engine for playing audio, or video, etc. can be modified for transmission-based billing according to these methods. In addition, the methods and systems described herein are extendable to content that can be streamed, such as text, video, audio, graphics, etc.

The PBBS dynamically provides billing tracking code and associated communication support, referred to collectively herein as "billing tracking code," to enable application, and service providers or other content providers (typically carriers) to automatically track billing information on a configurable, logical packet level, based upon the amount of data sent and/or received by the application or service. The data tracked is associated with a particular user, thus allowing the content provider to bill on a per-user, per-application basis based upon actual volume of use. The content provider (used herein to refer generally to any type of content provider) can subsequently implement a variety of billing policies at the application or user level, which can change over time. For example, an application provider may desire to charge lower fees for popular applications when they are heavily used by a subscriber, for example, as measured by the quantity of packet data received/sent by those applications for the particular subscriber. As another example, a service provider may desire to implement a promotion for a new application, charging less per transmission volume for that particular application in comparison to a normal fee.

Figure 1 is an overview block diagram of an example transmission-based billing system. In Figure 1, the Packet-Based Billing System 100 comprises several components: a content (code) modifier 103 for modifying requested content to contain billing tracking code; a configuration data repository 105 for storing the transmission and billing tracking code; a proxy server and/or billing server 104 for receiving and collecting billing data from content that is downloaded to target (client) devices; a

billing data repository 106 for storing the collected billing data and business rules for collecting billing information; and an accounting program 107 for reading the collected billing data and generating customer billing records 108. As will be described in further detail below, the PBBS components are typically integrated into a system that provides content to target devices. In operation, a client device, such as personal computer 101 or wireless handset 102, requests content, such as an application, from a system that provides the content. The content may be requested from a system that is connected via a wireless network to a wireless device, such as handset 102, or from a system that is connected via a wired network to a wired device, such as personal computer 101. As part of the process of requesting content, after the system determines and retrieves the requested content, the content modifier 103 analyzes the content and consults the configuration data repository 105 to determine what type billing tracking code needs to be inserted into the requested content and transparently modifies the content accordingly. The modified content is then returned to the client device 101/102 for downloading. At subsequent times, when the downloaded modified content is executed on the client device 101/102, the billing tracking code is automatically executed to collect and post billing data to the proxy/billing server 104. The billing server 104 collects and stores the received billing data according to the business rules in billing data repository 106. The accounting program 107 can then retrieve the collected billing data to generate the customer billing records 108. Typically, the accounting program is specifically designed to accommodate the needs of a service responsible for the billing, such as a carrier in a wireless networking environment.

Example billing data includes the number of bytes sent and/or received, a time stamp, an application identifier, user identifier (sent automatically by the carrier infrastructure when billing data is sent over a wireless network), a security key, a transaction identifier, and a retry expiration indicator. The transaction identifier is typically used to identify individual/different billing tracking "events." The retry expiration indicator indicates how long a client device should continue to re-post the same billing data, when the client device realizes a post operation has failed. Similarly, a proxy/billing server uses the retry expiration indicator to determine how long a transaction identifier is valid, in order to detect duplicate received postings.

In one embodiment, the PBBS content modifier (*e.g.*, content (code) modifier 103 of Figure 1) instruments packet-based billing tracking code into the requested content to accomplish the techniques of the present invention. Instrumentation, as used herein, is a non-intrusive means for modifying content (*e.g.*, an application) to include additional code, which in this case is billing tracking and transmission code. In one embodiment, network calls resident in the content are detected by the PBBS code modifier and replaced by transmission-based billing specific billing tracking code, which computes and collects the packet-based billing data and invokes the originally specified network calls. Instrumentation in this sense is used as a "hook" before or after a network call as a method for intercepting the network call. Because different content in different environments will use a variety of network calls and because different providers having different billing policies, the PBBS code modifier uses a configuration data repository (such as configuration data repository 105 in Figure 1) to determine which replacement network call to instrument into the content. The replacement calls may store the collected billing data locally in temporary storage on the target device and then post the collected data when the content finishes execution (for example, when an application exits). Alternatively, the collected data may be posted upon each network call. In addition, the collected data may be stored in permanent storage on the client device so that it may be posted after cycling power on the device. One skilled in the art will recognize that different scenarios for when data is posted are possible and contemplated to work with these techniques.

Figure 2 is an example block diagram of content instrumented with billing tracking code as performed by a content (code) modifier of a Packet-Based Billing System. Code, for example, an application 210, is sent to the code modifier 201 for analysis and instrumentation. The calling sequence of routines within application 210 is shown schematically (after undergoing a calling sequence analysis). According to the illustrated sequence, a "startup()" routine 211 is called, followed by a network call 212, followed by an "end()" routine 213 for application cleanup, followed by an exit call 214 to end application execution. The content modifier 201, looks up the network call in the configuration data repository 202, and determines and retrieves a corresponding network call 222 which contains code for implementing transmission-based billing and tracking. The corresponding network call 222 is implemented to invoke the original network call 212, thus it provides a "hook" to the original network call 212. The content modifier 201 also consults the configuration data repository 202 to determine configuration parameters for billing such as how frequently billing information is to be posted (for example, after a particular time or amount of data has been transmitted) and the network address of the proxy server to which the billing data is to be posted. The content modifier 201 then instruments the application with the retrieved network call 222 and code for posting billing data, as indicated by the determined configuration parameters. A schematic of an example modified application 220 is shown with a calling sequence after instrumentation has been performed. In the

modified application 220, the "startup()" routine 211 is called, now followed by the instrumented network call 222, which is followed by the original network call 212, followed by the "end()" routine 213. The content modifier also instruments code 223 to post billing data to the proxy/billing server according to the determined configuration parameters. This posting code 223 is shown in the sequence between the "end()" routine 213 and the exit call 214. Having code that posts billing data at the end of the application, even if billing data is posted earlier, attempts to retry posting any previously failed postings. In one embodiment, if the final posting call 223 fails, the billing data is simply lost and ignored. Other embodiments may choose to more permanently store the billing data and retry posting when the application is executed again. The retry expiration indicator can be used to prevent redundant billing events. Specifically, the client device determines, based on the timestamp, whether a retry period has expired, or whether the posting code should attempt to re-post the same billing data. One skilled in the art will recognize that many such variations are possible. In addition to the instrumented billing tracking code, in some embodiments, the content modifier 201 adds a security key 230 to the modified application in order to insure that billing data for the application, once downloaded cannot be misappropriated by sending fake billing data.

In other embodiments, as described in detail in Appendix E, the billing tracking code is incorporated into content by other means, including intrusive and non-intrusive means. For example, a specification for packet-based billing tracking code can be provided to content providers which can modify their content to explicitly include such tracking. In a second example, an Application Programming Interface (an "API") can provide library of functions that can be invoked by content providers at appropriate points in the content to provide packet-based billing tracking and communication. In yet a third example, the network driver software of the target device, such as a wireless device, can be modified to include the packet-based billing tracking code (through specification or library mechanisms). In all of these other embodiments, either the content provider or the device driver manufacturer needs to be made aware of the billing tracking code and communication techniques; thus providing a more intrusive means of incorporating the billing support.

The PBBS components and functionality may be integrated into and dispersed over different components in a network environment, such as an application provisioning system. In that scenario, provisioning components then interact to determine and insert billing tracking code into the content, receive data from the billing tracking code, and process the tracked data in conjunction with indicated billing policies to generate billing records. More specifically, when an application is requested by a subscriber, the application is provisioned by the application provisioning system for the requesting device and is downloaded to the requesting device, with the appropriate billing code instrumented into the application. Provisioning, as used herein, is the customizing and distributing of content for a particular use, for example, for use on a particular kind of subscriber device by a particular customer. An example application provisioning system, referred to as the Mobile Application System (MAS), can be used with the PBBS discussed herein. Appendix D describes such a system in detail, including techniques for instrumenting code into applications, customizing such applications, and distributed them to, especially, wireless devices. The MAS is a collection of interoperating server components that work individually and together in a secure fashion to provide applications, resources, and other content to mobile and wired subscriber devices.

Figure 3 is a block diagram of an example transmission-based billing system implemented within an example Mobile Application System environment. The Mobile Application System (MAS) shown is a provisioning system connected over a wireless connection to a wireless handset 310. The components of the MAS include, among other components, the Provisioning and Deployment Managers 302, which contain as part of their function, content scanner, analyzer, and instrumentor capabilities 303; a Billing Manager 305; a Configuration data repository 304; and an Accounting Program 306. The instrumentor capabilities 303 of the Provisioning and Deployment Managers 302 provide the code modifier functions of the PBBS. The Billing Manager 305 can incorporate the role of a proxy server for collecting posted billing data. The wireless handset 310 includes typically flash memory 311, or other type of local, semi-permanent storage for holding billing data as it is collected. The wireless handset 310 is also connected over a wireless connection to a public network, such as Internet 320. For the purposes of ease in description is assumed that the handset 310 is capable of addressing servers on the public network 320 directly and separately from posting billing data to the MAS Billing Manager 305; however, this assumption is not necessary to accomplish the techniques described herein. In particular, if a client device is unable to directly address multiple servers, then a proxy server can be provided that implements store and forward capabilities for all received network packets. In this scenario, the proxy server receives a packet and determines its intended destination (as well as receiving the billing data); retrieves the billing data from the packet, and forwards the original packet data to its intended destination.

Although the techniques of the PBBS are generally applicable to any type of client wireless device, one skilled in the art will recognize that terms such as subscriber device, client device, phone, handheld, etc., are used interchangeably to indicate any type of subscriber device that is capable of operating with the PBBS. Also, terms may have alternate spellings which may or may not be explicitly mentioned. For example, byte-code may be also indicated as "bytecode" or "Bytecode," and one skilled in the art will recognize that all such variations of terms are intended to be included. In addition, example embodiments described herein provide applications, tools, data structures and other support to implement a transmission-based billing system over one or more networks. One skilled in the art will recognize that other embodiments of the methods and systems of the present invention may be used for many other purposes, including instrumenting transmission-based billing support into software and other content over non-wireless networks, such as the Internet, to non-wireless subscriber devices, such as a personal computer, a docked wireless handset, telephones with Internet connectivity, or customer kiosks, for example, within airports or shopping malls. In addition, although this description primarily refers to content in the form of applications, services, and resources, one skilled in the art will recognize that the content may contain text, graphics, audio, and video. Also, in the following description, numerous specific details are set forth, such as data formats and code flows, etc., in order to provide a thorough understanding of the techniques of the methods and systems of the present invention. One skilled in the art will recognize, however, that the present invention also can be practiced without some of the specific details described herein, or with other specific details, such as changes with respect to the ordering of the code flow.

In addition, the techniques of the PBBS can be extended to operate with streaming content. Specifically, when streaming content (such as text, audio, video, graphics, etc.) is requested by a client device, the request indicates an amount of content to download. The content modifier, instead of inserting billing and tracking code into the content, generates billing events directly and sends them to a proxy/billing server.

Also, one skilled in the art will recognize that the techniques of the present invention can be used for other uses in which determining and tracking the amount of data sent and received is valuable, other than for generating billing data. For example, an additional use for the techniques of the present invention relates to the routing of network packets and requests. Specifically, the same techniques for tracking billing data based upon the amount of data sent and received between content and the network can be used by a proxy server (or other component or system) to decide how and where to route network packets based upon a routing policy. The "billing" data posted includes information that identifies the user, the application and the amount of data being transmitted, which can be used by a routing system to route traffic. For example, an application server may wish to distribute network traffic in a particular manner or reserve particular servers for application/user combinations that are heavily trafficked; thus, providing a type of load balancing.

Figure 4 is an example block diagram of a general-purpose computer system and a client device for practicing embodiments of the transmission-based billing system. The computer environment of Figure 4 comprises a client (subscriber) device 401 and a general purpose computer system 420, which communicate via a network 410. Each block may represent one or more such blocks as appropriate to a specific embodiment, or be combined with other blocks, and each may reside in separate physical locations.

The client device 401 comprises a computer memory ("memory") 402, a display 404, Input/Output Devices 403, and a Central Processing Unit ("CPU") 405. Modified Content 406, for example an executable application, is shown residing in memory 402 with other downloaded applications 407 and a data repository for temporary storage of billing data 408. The Modified Content 406 preferably executes on CPU 405 and executes the inserted billing tracking code, as described in previous figures, to track transmission data and to communicate the billing data to a proxy/billing server across the network 410.

The general-purpose computer system 420 may comprise one or more server and/or client computing systems and may span distributed locations. In one embodiment, wherein the PBBS is integrated into an application provisioning system such as a MAS, the MAS is implemented using Java 2 Enterprise Edition (J2EE) and executes on a general-purpose computer system that provides a J2EE compliant application server. According this embodiment, the MAS is designed and coded using a J2EE multi-tier application architecture, which supports a web tier, business tier, and a database tier on the server side. Thus, general purpose computer system 420 represents one or more servers capable of running one or more components and/or data repositories of the MAS and the PBBS.

As shown, general purpose computer system 420 comprises a CPU 423, a memory 421, and optionally a display 422 and Input/Output Devices 424. The components of the PBBS 430 are shown residing in memory 421, and preferably execute on one or more CPUs 423. Other data repositories and other programs (not shown) also reside in memory 421, and preferably execute on one or more CPUs 423.

In a typical embodiment, the PBBS 430 includes a Content Modifier 425, Data Repositories 427 and 428 for storing transmission and billing tracking code, billing data and business rules, Billing Server 426 (which is shown acting as the proxy and billing server), and Accounting Program 429. As described earlier, the PBBS may include other data repositories and components depending upon the needs of and integration with the carrier or other host systems. Other components, which are part of the application provisioning system, are also present in the memory 421, but not shown, such as the provisioning and deployment components and a local applications store. As mentioned, applications are provisioned and instrumented with the billing tracking code by the Content Modifier 425 before downloading to the client device 401.

One skilled in the art will recognize that the PBBS 430 may be implemented in a distributed environment that is comprised of multiple, even heterogeneous, computer systems and networks. For example, in one embodiment, the Content Modifier 425 and the Billing Server 426 are located in physically different computer systems. In another embodiment, various components of the PBBS 430 are hosted each on separate server machines and may be remotely located from the data repositories 427 and 428. In addition, under some scenarios, the Accounting Program 429 may be hosted within a carrier's infrastructure and be completely separated from the PBBS. Different configurations and locations of programs and data are contemplated for use with techniques of the present invention.

In an example embodiment, components of the PBBS 421 are implemented (as part of the MAS) using a J2EE multi-tier application platform, as described in detail in *Java™ 2 Platform, Enterprise Edition Specification, Version 1.2*, Sun Microsystems, 1999. The Content Modifier 425 is typically part of the MAS Provisioning and Deployment Managers (as shown in Figure 3). The Billing Manager 426 is a component of the MAS, enhanced to perform the various capabilities associated with transmission based billing. The data repositories 427 and 428 for storing the code to be instrumented, business rules, and billing data may be part of the Configuration Manager of the MAS (see Configuration Data Repository 304 of Figure 3) or may be implemented as separate data repositories, depending upon security needs, the location of the Accounting Program 429, etc. Figures 5-13 describe various example embodiments of the specific routines implemented by each of these components to achieve the functionality described with reference to Figures 1-3. In example embodiments, these components may execute concurrently and asynchronously; thus, the components may communicate using well-known message passing techniques. One skilled in the art will recognize that equivalent synchronous embodiments are also supportable by a PBBS implementation. Also, one skilled in the art will recognize that other steps could be implemented for each routine, and in different orders, and in different routines, yet still achieve the functions of the PBBS.

As described with respect to Figure 1, client (subscriber) devices can request an application from an application provisioning system, such as a Mobile Application System. Using the MAS, the application may be pre-provisioned for the device and subscriber and stored locally within the MAS (termed "walled garden provisioning") or may be provisioned on the fly when an application is requested, for example, by browsing a site over the Internet (termed "open provisioning"). Figure 5 is a block diagram of an example procedure for submitting an application to a Mobile Application System for use with transmission-based billing. In Figure 5, a content provider 501, such as a third party application provider or a carrier, submits an application to the provisioning system, here shown as MAS 502. The MAS 502 stores the application (either as raw data or pre-provisioned) in local storage 503. The content provider 501 also provides billing related business rules, which are stored appropriately by the MAS 502 in Billing Data Repository 504. These rules indicate billing related information such as the frequency or interval for posting the billing data, the address of the proxy server to send the billing data to, the size of a logical packet, and billing charge information associated with the logical packet size. One skilled in the art will recognize that other billing related business rules that are application- or user-specific may also be stored as needed.

Figure 6 is an example block diagram that illustrates an overall process for transparently delivering an application that supports transmission-based billing to a client device using a Mobile Application System. The client device 601 requests an application from the MAS 602 using the command handler 602, which processes requests for the MAS. The command handler 602, is responsible for distributing the application request to the appropriate component of the MAS, such as the MAS Provisioning and Deployment Managers (not shown). These components, which also contain the Content Modifier functionality, determine whether an application that corresponds to the requested application already exists in the cache 603, or whether an application needs to be provisioned for deployment. As part of the provisioning process, billing tracking code is instrumented into application 605 and a security key 606 is added, to generate modified application 604. Even if a provisioned application with instrumented billing tracking code is available in the cache 603, it may be necessary (depending upon the technique used) to generate a security key. The security key 606 is preferably generated and stored in a secure data repository 607 along with an

associated subscriber identifier and application identifier. Any mechanism for generating a security key that is uniquely associated with a subscriber and an application may be incorporated and used with the techniques of the present invention. One mechanism is to generate a n-bit random number and combine it in some way with a unique application identifier and a unique subscriber identifier. This mechanism allows a single security key to be reused for more than one application/subscriber combination, because the key is uniquely tied to each application/subscriber combination. Thus, modified content (which includes the billing tracking code and security key) can be cached, hence allowing faster downloads of content. Alternatively, a unique security key can be associated with each application/subscriber version of content. The purpose of the security key is to generate a number that can be later recognized as uniquely belonging to a particular subscriber and application when billing data is posted to the proxy/billing server for collection and processing. The modified application 604 is optionally stored in the cache 603 and returned through the command handler 602 to the requesting client device 601. It may be desirable to store the modified application 604 in the cache 603 for a short period of time in case the request failed to download properly and the client device 601 retries the request for the same application. Note, however, that in one embodiment, the security key that is added to the application is associated with a particular subscriber. In that scenario, storing the modified application 604 with a security key in the cache 603 doesn't make sense for deploying to a different application/subscriber combination.

Figure 7 is an example flow diagram of a routine for modifying an application to support transmission-based billing. This routine is typically executed as part of the provisioning and deployment process (see Content Modifier 201 in Figure 2). In summary, the routine determines or generates a version of the designated application with instrumented billing tracking code, determines or generates an appropriate security key which is inserted into the application, and returns the modified application to the requestor. Specifically, in step 701, the routine receives an application request with a designated application as a parameter. In step 702, the routine determines whether an instrumented application is already available (for example, is stored in cache 603 in Figure 6), and, if so, continues in step 705, else continues in step 703. In step 703, the routine analyzes the code flow of the application and instruments the billing tracking code into the application. This process is described further below with respect to Figure 8. In step 704, the routine stores the instrumented application, for example, in an application cache, and continues in step 707. In step 705, after determining that an instrumented application is already available, the routine retrieves the instrumented application from local storage, *e.g.*, the cache, and continues in step 706. In step 706, the routine determines whether the retrieved application already has a security key attached or associated with it, and, if so, continues in step 710 to retrieve the security key (or modified application with the security key), else continues in step 707. It may be desirable to keep an attached security key with the instrumented application for a limited period of time to limit potential misappropriations. In step 707, the routine generates a new security key for the application. Any appropriate security mechanism may be utilized, including the one described with respect to Figure 6. In step 708, the routine stores the newly generated security key in a secure data repository. In step 709, the routine instruments the new security key into the application. In step 711, the routine forwards the modified application which now includes the instrumented billing tracking code and the security key to the requestor, and ends processing.

Figure 8 is an example flow diagram of a routine for analyzing and instrumenting an application with support for transmission-based billing. This routine transforms an application to a modified application that contains support for transmission-based billing as illustrated in Figure 2. In step 801, the routine scans and analyzes the structure and calling sequence flow of the application, preferably at the byte-code (sometimes termed "binary application") level, to understand the data structures (package, class, method, and field definitions) and the calling sequences. As part of this deconstruction/decoding process, the routine detects all of the APIs present in the application and identifies any network calls. As a result of this analysis, in step 802, the routine identifies what code calls the network calls, hence where the calls are located within the application. When the applications are coded in Java, then this analysis can be performed at a byte-code (or binary) level of the program, with no need to insert analysis generating code at the source level. (The byte-code level refers to an "intermediate" level of binary code, which is interpreted by an "engine," "byte-code" interpreter, or "virtual machine," etc. in order to execute.) One skilled in the art will recognize that, other embodiments can be implemented for other languages and content, providing the data structures and calling sequence can be detected and analyzed. Java and .NET applications, in particular, are inherently well suited to such a process because they are instruction driven – different byte codes are used to indicate different language elements. Other intermediate code languages can be similarly analyzed. In step 803, the routine determines proxy network calls that correspond to the located network calls within the application. These proxy calls are typically determined from a configuration data repository which associates various network calls with particular

devices and protocols. Appendix A includes example pseudo-code for mapping located network calls to proxy network calls. In step 804, the routine determines specific business rules that apply to this application as specified, for example, by a carrier. As described earlier, these rules may define the interval/frequency for billing data to be posted to the proxy/billing server, the logical size of a packet to be used for charging, and the charge associated with a packet. An extensive set of rules may be specified on an application or on a per-user basis. One skilled in the art will recognize that other business rules can be specified as appropriate and can be changed over time. For example, in some systems, the application programmer may be allowed to provide default charge information which can then be modified by a standard multiplier by the carrier, hence the PBBS. In step 805, the routine replaces the identified network calls in the application with the proxy network calls, as illustrated in Figure 2. In step 806, the routine adds a final proxy network call to post billing data at the end of application processing, as illustrated in Figure 2. This call is typically added even if the prior proxy network call posted billing data to the proxy/billing server in case a prior call failed. In a situation where the prior proxy network calls collect the data locally on the client device, this final call to post data communicates the accumulated set of billing data.

As described with respect to Figure 1, when modified content is executed (processed) on a client device, the billing tracking code is activated, collects the data, and automatically posts it to a proxy/billing server associated with the PBBS. Figure 9 is an example block diagram that illustrates an overall process for communicating billing data based upon network transmission data on a per application basis. Although for the purposes of example, the components are presumed to be part of an application provisioning system, one skilled in the art will recognize that, as described earlier, they can be integrated into any content delivery system capable of performing the functions of the PBBS. In Figure 9, a client device 901 posts billing data (through network packets) to a proxy server 902, as a result of the posting code previously instrumented into the application (see Figures 2 and 8). Two different types of proxy servers may be implemented depending upon the capabilities of the devices. In particular, some devices can send network packets directly to more than one server. In that case, packets 909 destined for other servers can be sent directly to them, while billing data can be sent directly to the proxy server 902. (Such a setup is sometimes referred to as a compute and log approach.) Other devices are able to send network packets to only one server. In this scenario, the proxy server 902 acts as a store and forward proxy server and distributes network packets to their intended destinations, processing only the packets with the posted billing data. The proxy server 902, if a separate server, collects the posted data and forwards it as appropriate to a billing server 903. In some embodiments, the proxy server and billing server are combined. In some situations, however, for example due to security concerns, it may be desirable to separate the proxy and billing servers. As another example, the billing server may already exist and have particular data collection protocol to which the proxy server interfaces. Once the collected billing data is received, the billing server 903 determines whether the security key sent with the billing data is legitimate by comparing it with the expected security key in security data repository 905 for that particular application and that user. If the security key is legitimate, the billing server 903 stores the raw billing data in a data repository 906. The billing server 903 may optionally post-process the billing data according to business rules stored, for example, in repository 906. One skilled in the art will recognize that the data repositories are separated as shown for mere illustration, and that other combinations can be used, such as a single data repository. In addition, the proxy server 902 may store the raw billing data directly, to be processed asynchronously by the billing server 903. The accounting program 904 retrieves the billing data (raw or post-processed) from the data repository 906 and optionally uses overriding business rules, for example, as stored in data repository 907, to further process the billing data and to generate call (customer) data records 908. Overriding business rules may include, for example, specific application or subscriber overrides, promotions, etc.

Figure 10 is an example flow diagram of a client routine for sending data to a network that has been modified to collect and transmit billing data. This routine illustrates example proxy network code that replaces some type of "send_data" network call. An example pseudo-code implementation of proxy network code that sends data across a network is included as Appendix B. The routine tracks and accumulates the amount of data being sent, and, when the accumulated amount corresponds to the business rule incorporated into the routine for the frequency/interval for posting the billing data, then the billing data is posted. Specifically, in step 1001, the routine determines the amount of data that is to be sent in the current packet and accumulates it (*e.g.*, in a variable "data_out"). In step 1002, the routine stores the amount of data in the current packet along with a timestamp, application identifier, and security key in local storage. In step 1003, the routine determines whether it is time to post the data (*e.g.*, whether the coded interval of time/frequency has passed for posting the data), and, if so, continues in step 1004 to post the billing data to the proxy server, else continues in step 1006. In step 1006, the routine determines whether the business rule for posting data based is count based (based upon the amount of data) and, if so, whether the count

is greater than the indicated rule for posting data. If so, then the routine continues in step 1005, else continues in step 1007. In step 1005, the routine resets the data amount counter and continues in step 1004 to post the billing data to the proxy server. In step 1007, the routine sends the data using the original network call coded into the application, and then returns.

Figure 11 is an example flow diagram of a client routine for receiving data from a network that has been modified to collect and transmit billing data. This routine illustrates example proxy network code that replaces some type of "receive_data" network call. An example pseudo-code implementation of proxy network code that receives data across a network is included as Appendix C. The routine intercepts the original "receive_data" network call, tracks and accumulates the amount of data being received, and, when the accumulated amount corresponds to the business rule incorporated into the routine for the frequency/interval for posting the billing data, then the billing data is posted. Specifically, in step 1101, the routine receives the data using the original network call and intercepts the return. Then, in step 1102, the return determines whether the original network call was successful, and, if so, continues in step 1103, else returns an error. In step 1103, the routine determines the amount of data that was received in the packet and accumulates it (*e.g.*, in a variable "data_in"). In step 1104, the routine stores the amount of data in the received packet along with a timestamp, application identifier, and security key in local storage. In step 1105, the routine determines whether it is time to post the data (*e.g.*, whether the coded interval of time/frequency has passed for posting the data), and, if so, continues in step 1106 to post the billing data to the proxy server, else continues in step 1107. In step 1107, the routine determines whether the business rule for posting data based is count based (based upon the amount of data) and, if so, whether the count is greater than the indicated rule for posting data. If so, then the routine continues in step 1108, else returns. In step 1108, the routine resets the data amount counter and continues in step 1106 to post the billing data to the proxy server, and then returns.

Figure 12 is an example flow diagram of a server routine for processing posted transmission-based billing data. This routine may be performed, for example, by a billing server, such as billing server 903 in Figure 9 or by a proxy server, such as proxy server 902 in Figure 9. One skilled in the art will recognize that the steps included here are merely illustrative, and that different steps may be substituted for or combined with these steps, depending upon the arrangement and integration of the functions of the PBBS in a surrounding environment. In step 1201, the routine receives the posted billing data from a client device. In step 1202, the routine extracts the security key, application identifier, and user identifier from the billing data. The routine, in step 1203, then compares this information with the application identifier and user identifier associated with that security key in a security key data repository table. In step 1204, if the security key information matches, the routine continues in step 1206, else the billing data is discarded in step 1205, and the routine returns. In step 1206, the routine stores the billing data (or forwards the billing data to a billing server). In embodiments in which devices are restricted to communication with one server system, then in step 1207, the routine detects and forwards data packets designated for other servers, and then returns.

Figure 13 is an example flow diagram of steps for generating billing charges based upon transmission-based billing data. This routine may be performed, for example, by an accounting program, such as Accounting Program 904 in Figure 9. The steps shown here are generally applicable to processing billing records; however, depending upon the specifics of the carrier or other content provider that is determining the billing rates, applicability and the format of the customer (call) data records, various additional or different steps may be included. In step 1301, the routine retrieves the transmission-based billing data. In step 1302, the routine determines the applicable business rules for the indicated application identifier and user identifier. In step 1303, the routine determines whether there are any overriding policies/business rules, for example, promotions, discounts, etc., and, if so, continues in step 1304 to determine the overriding rules, else continues in step 1305. In step 1305, the routine applies any determined business rules and generates call data records. The format of these records is highly dependent upon any billing system that the transmission-based billing data is being integrated into, for example, an already existing billing system within a wireless carrier infrastructure.

From the foregoing it will be appreciated that, although specific embodiments of the invention have been described herein for purposes of illustration, various modifications may be made without deviating from the spirit and scope of the invention. For example, one skilled in the art will recognize that the methods and systems discussed herein are applicable to content provisioning systems and transmission-based billing system across any network, wired or wireless, or even a plurality of such networks. One skilled in the art will also recognize that the methods and systems discussed herein are applicable to differing protocols, communication media (optical, wireless, cable, etc.) and subscriber devices (such as wireless handsets, electronic organizers, personal digital assistants, portable email machines, game

machines, pagers, navigation devices such as GPS receivers, etc.). Aspects of the invention can be modified, if necessary, to employ methods, systems and concepts of these various patents, applications and publications to provide yet further embodiments of the invention. In addition, those skilled in the art will understand how to make changes and modifications to the methods and systems described to meet their specific requirements or conditions.

APPENDIX A- E

# Packet Level Billing

For Network Applications in A Wireless Environment

4thpass
Always Connected

Confidential

## Table of Contents

## Definitions

### Application Types

For the purpose of this document, the applications that could run on a wireless device are divided into two categories:

**Non-Network Applications:**

These types of applications do not communicate over network when active. These applications neither send any data to network, not they receive any from the network.

Examples: address book, calculator, text editor, etc.

**Network Applications:**

These types of applications use some sort of network communication to access other applications/devices/services on the network when active. These applications will send/receive messages on network

Example: browser, email client, chat program, calendar, etc.

### Server

For the purpose of the document, a server is generally software (which may not necessarily map to server hardware) that offers specific data or services to clients. Functionality of a given server is used to identify it – *e.g.* a Billing Server (generates and/or logs billing specific data), a Proxy Server (acts as a proxy in communicating with other servers), etc.

### Client

For purpose of this document, a "client" is used to refer to a network application running on a wireless device, communicating to a server in a traditional "client - server" scenario.

## Background

Typically, a wireless device user is charged when she/he uses airtime or when the user downloads any application or any other data to the device, depending on the carrier policies. All current breeds of solutions provide means to bill a user for airtime and total amount of data downloaded on the device. No solutions are currently available at software level or hardware level to allow for a carrier to charge a customer for using a **particular network application or portion of an application** on wireless devices based on the data that the specific application transmits or receives.

## Abstract

4thpass proposes a set of solutions to determine data usage of a specific network application at data packet level (i.e. the data packets sent and received by the wireless device). These solutions will allow a carrier to specify billing policies taking into consideration not just the data sent and received but also the application in question.

The network applications use low level communication protocols like TCP/IP and UDP/IP and higher-level protocols like HTTP and WAP (Wireless Application Protocol). The proposed solutions are generic solutions and work with any communication protocol. The implementation of a particular solution is protocol-specific and application language-specific.

The different solutions proposed in this document require different types of support on the device (where the application resides) and on the billing server (where billing specific data is generated and/or logged). This document describes different ways to implement different solutions to achieve the goal of generating packet level information for each application so that a user can be billed based on the data they transmitted / received instead or in addition to airtime billing policies.

A complete solution for providing packet level billing involves a client-side solution and a server-side solution. Depending on the solution, the billing information is either generated by the client or the server. The billing server ultimately collects data size/data packet information based on which billing records are generated.

## 4thpass Billing Solutions for Network Applications

4thpass proposes the following categories of solutions for generating billing data depending on the data transmitted and received by an application while it is active on a device. Each category of solution provides different ways to implement the solution.

1. **"Writing to Specification" Solution:** A network application developer uses specifications provided by 4thpass, for writing her/his code. The specifications indicate how to write network communication part of code so that it generates appropriate billing information at the packet level.

2. **Library Solution:** A network application developer uses a specific set of Application Programming Interfaces (APIs) that generate billing information, provided by 4thpass in the form of a software library, to write her/his application. This solution is an extension of the previous solution.

3. **Device Software Modification Solution:** A device manufacturer modifies the network functionality software layer on the device to generate billing information.

4. **Instrumentation Solution:** Software scans and modifies an application automatically to use billing APIs that generate billing information transparently (without altering application functionality) before the application is installed on the device.

Broadly speaking, the solutions can be further divided into two categories, depending on whether an application developer has to write code in a specific way or using specific tools.

1. **Intrusive:** The first two categories of solutions described above – "Writing to Specification Solution" and "Library Solution," are intrusive solutions. An application developer has to code in a specific way or has to code using special Application Programming Interfaces (APIs) in order to generate packet level

billing information.

2. **Non-Intrusive:** The third and fourth categories described above – "Device Software Modification Solution" and "Instrumentation Solution," are non-intrusive solutions. A developer does not have to do anything special while writing her/his application. Instead, 4thpass provided tools take care of the necessary functionality of generating packet level billing automatically.

All categories – intrusive and non-intrusive, their pros and cons, and the server side of solutions specific to them, are discussed here in detail. Each category of solution, in turn proposes a different mechanism to achieve the required functionality.

## "Writing to Specification" Solution (Intrusive)

This solution requires 4thpass to provide detailed specifications to the application developers, describing ways in which application code is to be written.

The specifications provide information on how to write application code to generate required billing information. A developer has to write code in a specific format and a specific sequence, meeting specific guidelines so that the billing information is properly generated.

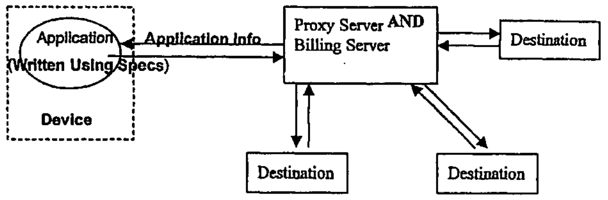
When writing to this specification, the following proxy-based mechanism can be used to generate billing information.

### Proxy Store and Forward Approach

Code for an application is written to communicate to a predefined proxy server. The data is sent to the proxy server in a pre-defined format (including the destination information and data that is to be forwarded). The proxy server reads the pre-defined data (stored in known format), extracts information from it that it requires to correctly communicate to the destination, and forwards the data to the destination.

The data that is forwarded to the destination denotes the proxy server as source, thus allowing reverse communication from destination. On receiving data from destination, the proxy server determines the original sender and forwards the received data to it.

While forwarding data back and forth, the transmitted and received data size/data packet information is logged, for example, by the proxy server (acting also as a billing server or forwarding it to a billing server) so that it can be later used in generating billing information.



## Library Solution (Intrusive)

The "Library Solution" is an extension of the "Write to Specification Solution." Instead of providing an application developer with specifications, a developer is provided with a software library, with predefined set of Application Programming Interfaces (APIs), that interact transparently with the proxy server / billing server. A developer still has to follow a specific set of guidelines in using the API, but the underlying specification is captured by means of APIs and the developer does not have to work extensively to support the specification.

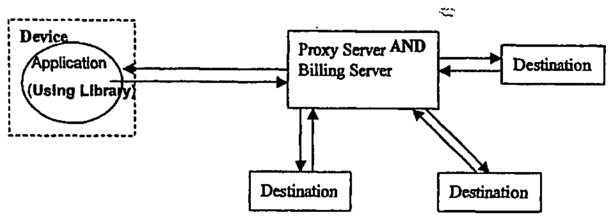
The Library solution can be used to allow support for two different kinds of implementations, by providing different and / or multiple libraries

### Proxy Store and Forward Approach

Code for an application is written using a predefined set of APIs. This approach is the same as that described under the "Writing to Specification solution," except that the library provided to an application developer implements and encapsulates the communication with the proxy server (between source and destination) using different APIs.

The pre-existing APIs allow a developer to develop her/his code without knowing intricacies of the specification and library code implementations.

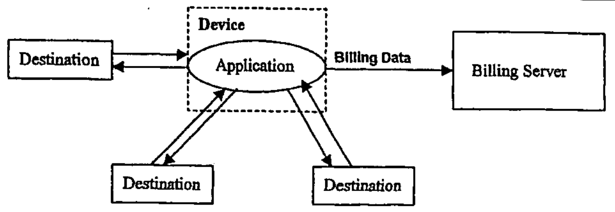
In this approach, the server generates (and logs) billing information.



## Compute and Log Approach

Another implementation of the library mechanism provides transparent logging / generating of the billing information. This approach is useful if the underlying network allows direct communication between devices. In this implementation, instead of using a proxy server for handling full bi-directional traffic for the client application, a billing server is only sent the billing data – i.e. packet size/number of packets, etc., and normal communication of the application data between the client and the server continues.

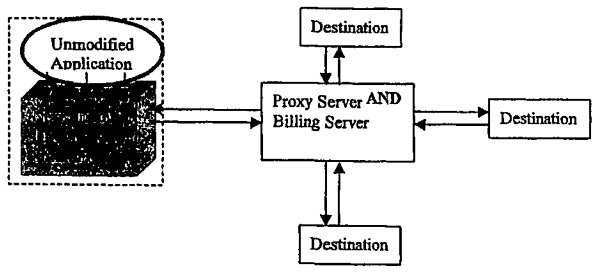
Specifically, the library API hides the implementation of the billing (logging) code, which now behaves differently than "proxy store and forward approach". Instead of sending the data to be transmitted to the proxy server (and receiving in turn data back from proxy server), the library API executing in the application determines the data size/packet numbers and sends this information to the billing server, transparently from the application program. The communication with the destination device or any external server is carried out directly.



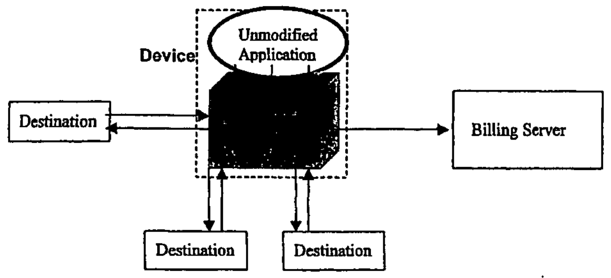
## Device Software Modification Solution (Non-Intrusive)

This approach requires 4thpass to work with the networking software driver (the software on the device responsible for providing networking connectivity) developer of a particular device to provide packet level billing. Using this approach, 4thpass provides a special piece of software that is integrated with the networking software driver. The special piece of software provides different functionality depending on whether a proxy approach or "compute and log" approach is taken. These approaches are the same as those described in the previous two sections, except that the network application developer does not have to write code in any specific way or follow any specifications. Instead, the networking communication driver code is responsible for providing billing specific functionality.

### Proxy store and forward Approach



### Compute and Log Approach



## Instrumentation Solution (Non-Intrusive)

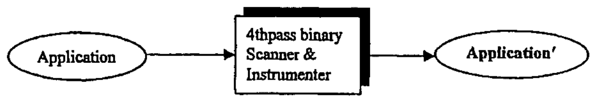
The instrumentation solution is a *hybrid* of the "library" and "writing to specification" solutions. Using this approach, APIs that are written to the billing specification are added to the network application, but not by the application developer.

Specifically, 4thpass provides automated software tools, which can read the network application and modify its binary (*e.g.*, byte-code in J2ME) image to include the appropriate APIs. The 4thpass software tools support instrumentation that can modify a binary image (*e.g.*, byte-code in J2ME) on the fly by adding, modifying or deleting existing binary code (*e.g.*, in byte-code format in J2ME).

Thus, by automatically introducing the instrumented code, the network program can satisfy any specification and provide billing solutions as required.

All the server side implementations discussed in previous sections (*e.g.*, proxy store and forward, and compute and log), apply. Just before installing the network application on

the device, the network application is run through 4thpass binary Scanner and Instrumenter. The Scanner determines the networking APIs used and automatically replaces them with the newer networking APIs. The newer networking APIs provide same functionality as the original ones, except that they also generate billing specific information.



Once the revised application runs (Application'), it uses the underlying networking APIs but these APIs provide the extended functionality to generate/log appropriate billing information.

In above picture **Application** represents the pre-4thpass-processed version of the network application, whereas **Application'** represents the post-4thpass-processed version. The server side support is the same as the "proxy store and forward" or "compute and log" approach (see, *e.g.*, the images described under the "Library Solution" sections).

## DNS based Server Side Implementation

In order to support packet level billing in clients in a one specific scenario, a DNS based solution is implemented on the server. This solution is useful to allow devices or programs from an outside network (such as those using public IPs) to access devices in an inside network using non-routable private IPs. In this manner, incoming data packets (to the network application) can be counted and the application billing information still can be generated.

The challenge is in providing a means for the devices or programs from the outside network to communicate to *internal* devices, since they are on very *different* networks and the data from a non-routable private IP generally cannot be sent to programs using public IP.

A 4thpass solution allows such communication by providing its own DNS server implementation on the carrier side, which works with a pool(s) of public IPs, which are dynamically mapped to the internal network private IPs. The DNS server manages the virtual communication link between two ends by providing Network Address Translation (NAT).

More information on is available in a separate paper, describing the technology to implement this solution.

## Comparison of Different Solutions

Each of the various solutions has its pros and cons. A specification-based solution is easier to manage, since it does not require distribution of library software. On the other hand, specifications are harder to understand and follow in comparison to a software library. Using a library, a network application developer calls the APIs without knowing what happens under the hood.

In comparison to both of these, modifying the networking software driver is to provide the same functionality is easier to implement and manage from an application developer standpoint. However, it becomes harder to manage different devices and a separate driver modification needs to be provided for each. Support for the different APIs in a scalable and error-proof way may be hard to maintain. Also it may be difficult to convince the numerous networking device driver authors to include 4thpass code.

In comparison to the above approaches, the Instrumentation approach is the most flexible and scalable. It provides the flexibility and advantages of previously described solutions. The software libraries of the APIs, (which are dynamically added to a network application after scanning them) are written to the specification and do not require a network application developer to understand how they are written. Nor does a developer need to know that the library APIs are to be added. Instead, upon scanning an application already written by an application developer, the 4thpass tools inspect the

application for usage of network APIs (irrespective of protocol). On detection of network APIs, the 4thpass tools replace the detected network APIs with a predefined set of APIs that implement the various specifications. This solution does not require any intervention on an application developer's part. Also, it is easier to manage its implementation because it is hidden from the developer (it may not even be made public). Moreover, it allows implementing an optimal solution, since the implementation can be changed and extended even while an older solution is in place. And last, but not least, it is less error prone because automatic steps are generally less error prone than manual steps.

Appendix A (connection_setup_proxycode)

```
/*****************************************************************************
  * INSTRUMENTATION METHODS
  * Here are example methods that will be instrumented in to application
  * (in pseudo-code)
  ******************************************************************************/

 /**
  * this is the replacement method for Connector.open(String)
  * Return the original connection bug add the connection
  * to the network connection list if it is network connection
  */
 public static Connection open(String name) throws IOException
 {
   Connection conn = Connector.open(name);

   if(name.startsWith("http://") || name.startsWith("socket://")) {
     _networkConnection.addElement(conn);
   }

   return conn;
 }

 /**
  * this is the replacement method for Connector.open(String, int)
  * Return the original connection bug add the connection
  * to the network connection list if it is network connection
  */
 public static Connection open(String name, int mode) throws IOException
 {
   Connection conn = Connector.open(name, mode);

   if(name.startsWith("http://") || name.startsWith("socket://")) {
     _networkConnection.addElement(conn);
   }

   return conn;
 }

 /**
  * this is the replacement method for Connector.open(String, int, boolean)
  * Return the original connection bug add the connection
  * to the network connection list if it is network connection
  */
 public static Connection open(String name, int mode, boolean timeouts)
  throws IOException
 {
   Connection conn = Connector.open(name, mode, timeouts);

   if(name.startsWith("http://") || name.startsWith("socket://")) {
     _networkConnection.addElement(conn);
   }

   return conn;
 }

 /**
  * this is the replacement method for the Connection.close() method.
  * This method will remove any network connection from the list
  * when it is close
  */
 public static void close(Connection conn) throws IOException
 {
```

Page 1

Appendix A (connection_setup_proxycode)

```
   _networkConnection.removeElement(conn);
   conn.close();
 }
```

Page 2

Appendix B (send_proxycode)

Note: The pseudo-code in this file are exerpts from the original code file
      that manages receiving and sending of the data.

```
/* Send data to the original remote host (as specified by original code)
 *   conn  -- Original connection created for communicating with host
 *   dgram -- The datagram that's to be sent.
 */

public static void send(DatagramConnection conn, Datagram dgram)
throws IOException
 {
   conn.send(dgram);
   // increase the total of bytes send
   incrementSentBytes(dgram.getLength());
 }


  /*
   * Increment the internal counter by number of bytes sent to the
   * remote host.
   */

  public static void incrementSentBytes(long bytes)
  {
    synchronized(_synchronizedObj) {
      _totalSentBytes += bytes;

      // we alway save billing info at the first time
      if((System.currentTimeMillis() - _time) > 60* 1000) {
        phaseOne();
        _time    = System.currentTimeMillis();
      }
    }
  }

  /**
   * This phase will save the heap info to rms
   */

  private static void phaseOne()
  {
    try {
      // check the caches
      if((_lastSentBytes == -1) && (_lastReceivedBytes == -1)){
        loadBillingInfo();
      }

      // save new billing info and update caches
      saveBillingInfo();

      // do autosend this code can be taken out depend on device
      if(System.currentTimeMillis() - _recordStoreTime > 24*60*60*1000) {
          phaseTwo();
      }
    }
    catch(Exception e){}
  }

  /**
   * Send billing info from rms to MAS server
   */
```

Page 1

Appendix B (send_proxycode)

```
public static void phaseTwo()
{
  try{
    // check the caches
    if((_lastSentBytes == -1) && (_lastReceivedBytes == -1)){
      loadBillingInfo();
    }

    // get total
    _lastSentBytes       += _totalSentBytes;
    _lastReceivedBytes   += _totalReceivedBytes;

    // send billing info
    autoSendBillingInfo();

    // successfull sending data so clear the rms
    clearRecordStore();
  }
  catch(Exception e){}
}

/**
 * This method will load the packet base billing record to the cache and
 * keep the billing info in the record store
 * This method will be used by the phaseOne and phaseTwo
 *
 */

private static void loadBillingInfo()
{
  try {
    synchronized (_synchronizedObj) {
      RecordStore recordStore =
         RecordStore.openRecordStore(RECORD_STORE_NAME, true);

      int id                = recordStore.getNextRecordID() - 1;
      byte [] record        = recordStore.getRecord(id);
      ByteArrayInputStream  bis = new ByteArrayInputStream(record);
      DataInputStream       dis = new DataInputStream(bis);

      // load the billing info
      _lastReceivedBytes          = dis.readLong();
      _lastSentBytes              = dis.readLong();
      _recordStoreTime            = dis.readLong();

      // close intput stream and record store
      // don't need to close ByteArrayInputStream
      recordStore.closeRecordStore();
    }
  }

  catch(Exception e) {

    // there is not thing in the record store.  Give an initialization data
    _lastReceivedBytes  = 0;
    _lastSentBytes      = 0;
    _recordStoreTime    = System.currentTimeMillis();
  }
}

/**
 * Save new billing info rms + heap then delete the old record and update
 * the caches
```

Page 2

```
 */

private static void saveBillingInfo()
      throws RecordStoreNotFoundException, RecordStoreException,
      IOException, RecordStoreFullException
{
  RecordStore recordStore = null;
  byte [] record = null;
  synchronized(_synchronizedObj) {
    // save the new info
    recordStore = RecordStore.openRecordStore(RECORD_STORE_NAME, true);

    ByteArrayOutputStream bos = new ByteArrayOutputStream(24);
    DataOutputStream  dos = new DataOutputStream(bos);

    // save new received bytes = last received + heap
    dos.writeLong(_lastReceivedBytes + _totalReceivedBytes);

    // save new sent bytes = last sent + heap
    dos.writeLong(_lastSentBytes + _totalSentBytes);

    // save time
    dos.writeLong(_recordStoreTime);

    // save
    record = bos.toByteArray();
    recordStore.addRecord(record, 0, record.length);

    // already saved new record so update last info
    _lastReceivedBytes  += _totalReceivedBytes;
    _lastSentBytes      += _totalSentBytes;


    // now clear the heap
    clearHeap();

    // already saved new record so delete the old one
    if(recordStore.getNumRecords() > 2) {
      recordStore.deleteRecord(recordStore.getNextRecordID() - 2);
    }

    // close output stream and record store
    // don't need to close ByteArrayOutputStream
    recordStore.closeRecordStore();
  }
}

/**
 * Send the packet base billing info to MAS.  After successful sending
 * billing record, clear all the record (heap and record store)
 * This method is used by the sendBillingInfo and saveBillingInfo
 */

private static boolean autoSendBillingInfo() throws IOException
{
  if(_lastReceivedBytes <=0 && _lastSentBytes <=0) {
    return true;
  }

  String es = "&";
  String eq = "=";
  if(ESCAPE_URL != 0) {
```

```
    es      = "%26";
    eq      = "%3D";
  }

  StringBuffer buff = new StringBuffer();
  // append url
  buff.append(MAS_PACKET_BASE_BILLING_URL);
  buff.append(es);

  // append total bytes sent
  buff.append("sent" + eq);
  buff.append(_lastSentBytes);
  buff.append(es);

  // append total bytes received
  buff.append("received" + eq);
  buff.append(_lastReceivedBytes);

  String request      = buff.toString();
  System.out.println(request);

  // try to send billing info for 3 times.
  int numOfRetry = 0;
  while(numOfRetry < 3) {
    try {
      HttpConnection conn = (HttpConnection) Connector.open(request);
      InputStream is  = conn.openInputStream();

      // close intput and connection
      is.close();
      conn.close();
      return true;
    }
    catch(Exception e) {
      numOfRetry++;
    }
  }

  // we don't need to check for response if the http connection fail, it
  // will through exception
  return false;
}


private static void clearRecordStore()
{
  synchronized(_synchronizedObj) {
    try{
      _lastReceivedBytes      = 0;
      _lastSentBytes          = 0;
      RecordStore recordStore =
            RecordStore.openRecordStore(RECORD_STORE_NAME, false);
      recordStore.deleteRecord(recordStore.getNextRecordID() - 1);
      recordStore.closeRecordStore();

      // reset the record store time
      _recordStoreTime = System.currentTimeMillis();
    }
    catch(Exception e){}
  }
}

// The local variables used in the above section of code
```

Appendix B (send_proxycode)

```
public static long      _totalSentBytes        = 0;
public static long      _lastSentBytes         = -1;
public static long      _totalReceivedBytes    = 0;
public static long      _lastReceivedBytes     = -1
public static Object   _synchronizedObj        = new Object();
public static long     _time                   = 0;
public static long     _recordStoreTime        = 0;
public static Vector   _networkConnection      = new Vector(3);

/////////////////////////////////////////////////////////////////////////
/////// Public Constructor

public BillingOutputStream(OutputStream out)
{
  _outputStream  = out;
}

/////////////////////////////////////////////////////////////////////////
/////// Public Members (Access Methods)

public void write(int b) throws IOException
{
  _outputStream.write(b);
  _totalSentBytes++;
  increment();
}

public void write(byte [] b) throws IOException
{
  _outputStream.write(b);
  _totalSentBytes += b.length;
  increment();
}

public void write(byte [] b, int off, int len) throws IOException
{
  _outputStream.write(b, off, len);
  _totalSentBytes += len;
  increment();
}

public void flush() throws IOException
{
  _outputStream.flush();
}

public void close() throws IOException
{
  // save the total sent bytes
  if(_totalSentBytes > 0) {
    PacketBaseBilling.incrementSentBytes(_totalSentBytes);
  }
  // reset the total sent bytes in case close() is called again
  _totalSentBytes   = 0;
  _outputStream.close();
}

private void increment()
{
  if(_totalSentBytes > PACKET) {
    PacketBaseBilling.incrementSentBytes(_totalSentBytes);
```

Appendix B (send_proxycode)

```
    _totalSentBytes = 0;
  }
}

/////////////////////////////////////////////////////////////////////////
/////// Private Fields
private OutputStream            _outputStream;
private long                    _totalSentBytes   = 0;
private final static int      PACKET              = 10;
```

Appendix C (receive_proxycode).txt

```
==============================================================================
 Note: The pseudo-code in this file are exerpts from the original code file
       that manages receiving and sending of the data.
==============================================================================

/*
 * This proxy code allows original connection to get data and then
 * increments the internal counter to record it.
 *   conn  -- Original connection created for communicating with host
 *   dgram -- The datagram that's to be sent.
 */

public static void receive(DatagramConnection conn, Datagram dgram)
throws IOException
{
  conn.receive(dgram);
  incrementReceivedBytes(dgram.getLength());
}


public static void incrementReceivedBytes(long bytes)
{
  synchronized(_synchronizedObj) {
    _totalReceivedBytes += bytes;

    // we alway save billing info at the first time
    if((System.currentTimeMillis() - _time) > 60*1000) {
      // save billing info
      phaseOne();
      _time   = System.currentTimeMillis();
    }
  }
}


/**
 * This phase will save the heap info to rms
 */

private static void phaseOne()
{
  try {
    // check the caches
    if((_lastSentBytes == -1) && (_lastReceivedBytes == -1)){
      loadBillingInfo();
    }

    // save new billing info and update caches
    saveBillingInfo();

    // do autosend this code can be taken out depend on device
    if(System.currentTimeMillis() - _recordStoreTime > 24*60*60*1000) {
        phaseTwo();
    }
  }
  catch(Exception e){}
}

/**
 * Send billing info from rms to MAS server
 */
public static void phaseTwo()
{
```

Page 1

Appendix C (receive_proxycode).txt

```
  try{
    // check the caches
    if((_lastSentBytes == -1) && (_lastReceivedBytes == -1)){
      loadBillingInfo();
    }

    // get total
    _lastSentBytes       += _totalSentBytes;
    _lastReceivedBytes   += _totalReceivedBytes;

    // send billing info
    autoSendBillingInfo();

    // successfull sending data so clear the rms
    clearRecordStore();
  }
  catch(Exception e){}
}

/**
 * This method will load the packet base billing record to the cache and
 * keep the billing info in the record store
 * This method will be used by the phaseOne and phaseTwo
 *
 */

private static void loadBillingInfo()
{
  try {
    synchronized (_synchronizedObj) {
      RecordStore recordStore =
          RecordStore.openRecordStore(RECORD_STORE_NAME, true);

      int id           = recordStore.getNextRecordID() - 1;
      byte [] record   = recordStore.getRecord(id);
      ByteArrayInputStream  bis = new ByteArrayInputStream(record);
      DataInputStream       dis = new DataInputStream(bis);

      // load the billing info
      _lastReceivedBytes        = dis.readLong();
      _lastSentBytes            = dis.readLong();
      _recordStoreTime          = dis.readLong();

      // close intput stream and record store
      // don't need to close ByteArrayInputStream
      recordStore.closeRecordStore();
    }
  }

  catch(Exception e) {

    // there is not thing in the record store.  Give an initialization data
    _lastReceivedBytes   = 0;
    _lastSentBytes       = 0;
    _recordStoreTime     = System.currentTimeMillis();
  }
}

/**
 * Save new billing info rms + heap then delete the old record and update
 * the caches
 */
```

Page 2

Appendix C (receive_proxycode).txt

```
private static void saveBillingInfo()
      throws RecordStoreNotFoundException, RecordStoreException,
      IOException, RecordStoreFullException
{
  RecordStore recordStore = null;
  byte [] record = null;
  synchronized(_synchronizedObj) {
    // save the new info
    recordStore = RecordStore.openRecordStore(RECORD_STORE_NAME, true);

    ByteArrayOutputStream bos = new ByteArrayOutputStream(24);
    DataOutputStream  dos = new DataOutputStream(bos);

    // save new received bytes = last received + heap
    dos.writeLong(_lastReceivedBytes + _totalReceivedBytes);

    // save new sent bytes = last sent + heap
    dos.writeLong(_lastSentBytes + _totalSentBytes);

    // save time
    dos.writeLong(_recordStoreTime);

    // save
    record = bos.toByteArray();
    recordStore.addRecord(record, 0, record.length);

    // already saved new record so update last info
    _lastReceivedBytes  += _totalReceivedBytes;
    _lastSentBytes      += _totalSentBytes;


    // now clear the heap
    clearHeap();

    // already saved new record so delete the old one
    if(recordStore.getNumRecords() > 2) {
      recordStore.deleteRecord(recordStore.getNextRecordID() - 2);
    }

    // close output stream and record store
    // don't need to close ByteArrayOutputStream
    recordStore.closeRecordStore();
  }
}


/**
 * Send the packet base billing info to MAS.  After successful sending
 * billing record, clear all the record (heap and record store)
 * This method is used by the sendBillingInfo and saveBillingInfo
 */

private static boolean autoSendBillingInfo() throws IOException
{
  if(_lastReceivedBytes <=0 && _lastSentBytes <=0) {
    return true;
  }

  String es = "&";
  String eq = "=";
  if(ESCAPE_URL != 0) {
    es      = "%26";
    eq      = "%3D";
```

Appendix C (receive_proxycode).txt

```
  }

  StringBuffer buff = new StringBuffer();
  // append url
  buff.append(MAS_PACKET_BASE_BILLING_URL);
  buff.append(es);

  // append total bytes sent
  buff.append("sent" + eq);
  buff.append(_lastSentBytes);
  buff.append(es);

  // append total bytes received
  buff.append("received" + eq);
  buff.append(_lastReceivedBytes);

  String request       = buff.toString();
  System.out.println(request);

  // try to send billing info for 3 times.
  int numOfRetry = 0;
  while(numOfRetry < 3) {
    try {
      HttpConnection conn = (HttpConnection) Connector.open(request);
      InputStream is  = conn.openInputStream();

      // close intput and connection
      is.close();
      conn.close();
      return true;
    }
    catch(Exception e) {
      numOfRetry++;
    }
  }

  // we don't need to check for response if the http connection fail, it
  // will through exception
  return false;
}


private static void clearRecordStore()
{
  synchronized(_synchronizedObj) {
    try{
      _lastReceivedBytes      = 0;
      _lastSentBytes          = 0;
      RecordStore recordStore =
            RecordStore.openRecordStore(RECORD_STORE_NAME, false);
      recordStore.deleteRecord(recordStore.getNextRecordID() - 1);
      recordStore.closeRecordStore();

      // reset the record store time
      _recordStoreTime = System.currentTimeMillis();
    }
    catch(Exception e){}
  }
}


// The local variables used in the above section of code
```

```
public static long      _totalSentBytes         = 0;
public static long      _lastSentBytes          = -1;
public static long      _totalReceivedBytes     = 0;
public static long      _lastReceivedBytes      = -1;
public static Object   _synchronizedObj         = new Object();
public static long     _time                    = 0;
public static long     _recordStoreTime         = 0;
public static Vector   _netWorkConnection       = new Vector(3);

public class BillingInputStream extends InputStream
{
  public BillingInputStream(InputStream is)
  {
    _is = is;
  }

  ///////////////////////////////////////////////////////////////////////////
  /////// Public Members (Access Methods)

  public int read() throws IOException
  {
    int r = _is.read();
    if(r != -1) {
      _totalReceivedBytes++;
    }
    increment();
    return r;
  }

  public int available() throws IOException
  {
    return _is.available();
  }

  public void mark(int readlimit)
  {
    _is.mark(readlimit);
  }

  public boolean markSupported()
  {
    return _is.markSupported();
  }

  public int read(byte [] b) throws IOException
  {
    int i = _is.read(b);
    _totalReceivedBytes += i;
    increment();
    return i;
  }

  public int read(byte [] b, int off, int len) throws IOException
  {
    int i = _is.read(b, off, len);
    _totalReceivedBytes += i;
    increment();
    return i;
  }

  public long skip(long n) throws IOException
  {
```

```
    return _is.skip(n);
  }

  public void reset() throws IOException
  {
    _is.reset();
  }

  public void close() throws IOException
  {
    // save the total received bytes
    if(_totalReceivedBytes > 0) {
      PacketBaseBilling.incrementReceivedBytes(_totalReceivedBytes);
    }

    // reset the total received bytes in case lose() is called again
    _totalReceivedBytes = 0;
    _is.close();
  }

  private void increment()
  {
    if(_totalReceivedBytes > PACKET) {
      PacketBaseBilling.incrementReceivedBytes(_totalReceivedBytes);
      _totalReceivedBytes = 0;
    }
  }

  ///////////////////////////////////////////////////////////////////////////
  /////// Private Fields

  private InputStream          _is;
  private long                 _totalReceivedBytes = 0;
  private final static int     PACKET              = 10;
}
```

MAINTAINING AND DISTRIBUTING WIRELESS APPLICATIONS

Embodiments of the present invention provide computer- and network-based methods and systems for maintaining and provisioning wireless applications. Provisioning, as it is discussed herein, is the customizing and distributing of content for a particular use, for example, for use on a particular kind of subscriber device by a particular customer. In an example embodiment, a Mobile Application System (MAS) is provided. The MAS is a collection of interoperating server components that work individually and together in a secure fashion to provide applications, resources, and other content to mobile subscriber devices. The MAS allows, for example, wireless devices, such as cellular phones and handset devices, to dynamically download new and updated applications from the MAS for use on their devices. Dynamic download capability significantly reduces time-to-market requirements for developers of wireless applications (content providers) and results in greater efficiencies in product support and marketing. Customers are able to quickly and conveniently update the operating software on their wireless devices and download popular applications (including games). With the MAS, customers are able to update their wireless handset devices directly from the network and thereby avoid the time-consuming experience of speaking to a customer service representative or visiting a local service center to update the software. The MAS also supports flexible billing scenarios, including subscription billing, which allows customers to subscribe to a particular service to receive only those resources or applications they desire.

Although the capabilities of the MAS are generally applicable to any type of client wireless device, one skilled in the art will recognize that terms such as subscriber device, client device, phone, handheld, etc., are used interchangeably to indicate any type of subscriber device that is capable of operating with the MAS. In addition, example embodiments described herein provide applications, tools, data structures and other support to implement maintaining and distributing wireless applications over one or more networks. One skilled in the art will recognize that other embodiments of the methods and systems of the present invention may be used for many other purposes, including maintaining and

distributing software and other content over non-wireless networks, such as the Internet, to non-wireless subscriber devices, such as a personal computer, a docked wireless handset, telephones with Internet connectivity, or customer kiosks, for example, within airports or shopping malls. In addition, although this description primarily refers to content in the form of applications and resources, one skilled in the art will recognize that the content may contain text, graphics, audio, and video. Also, in the following description, numerous specific details are set forth, such as data formats, user interface screen displays, code flows, menu options, etc., in order to provide a thorough understanding of the techniques of the methods and systems of the present invention. One skilled in the art will recognize, however, that the present invention also can be practiced without some of the specific details described herein, or with other specific details, such as changes with respect to the ordering of the code flow, or the specific features shown on the user interface screen displays.

Figure 1 is an example block diagram that illustrates how subscribers of wireless services request and download software applications from a Mobile Application System. The wireless environment in which the MAS operates includes subscriber devices 101 and 101b, wireless network 102 with transceiver 103, wireless carrier services 104, the MAS 105, and various content providers 106. Content providers 106 provide applications to the MAS 105, through or by permission of the carrier services 104. The applications are then verified, published, and provisioned to the subscriber devices 101 by the MAS 105 when requested. This type of provisioning is referred to herein as "walled-garden" provisioning, because applications that are provisioned and published in this fashion are known to the carrier and/or MAS infrastructure. Content providers 106 can also host applications that can be browsed by subscriber devices, which can be provisioned dynamically by the MAS 105. This type of provisioning is referred to herein as "open" provisioning, because it is not restricted to applications "known" within the confines of the MAS or carrier infrastructure. These distinctions are made for convenience of discussion alone, as the different types of provisioning share many similar functions. The MAS 105 also provides a multitude of tools for carriers, content providers, customer care representatives, and subscribers for customizing the applications, services, and billing scenarios available to specific subscribers or groups of subscribers.

In Figure 1, the subscriber devices 101 comprise electronic devices capable of communication over wireless network 102, such as wireless handsets, phones, electronic organizers, personal digital assistants, portable e-mail machines, game machines, pages, navigation devices, etc., whether or not existing presently. One or more of the subscriber devices 101 (also referred to as client devices) communicate across the wireless network 102 to the wireless carrier services 104, whose services the subscriber has arranged to use. The wireless network 102 has a transceiver 103, which is used to relay services to the subscriber devices 101 (and handle subscriber requests). One skilled in the art will recognize that a subscriber of wireless services may complement any or all of steps involved in requesting and downloading wireless applications across a wireless network by using an alternate network (such as the Internet) and by using devices having a larger form-factor, such as a personal computer 101b, which may offer easier-to-use interfaces for downloading applications. The transceiver 103 typically converts wireless communications to cable-based communications, and converts the cable-based communications to wireless communications, although one skilled in the art will appreciate that varying media and protocols may be used. Transceiver 103 typically communicates with the carrier services 104 across a cable-based medium using a carrier-specific communication protocol. The carrier-specific communication may use any protocol suitable for point-to-point communications such as Hypertext Transport Protocol (HTTP) and Wireless Application Protocol (WAP). The carrier services 104 provide services that are typical of a telephone central office including accounting, POTS ("plain old telephone service") and other telephone services (such as call forwarding, caller ID, voice mail, etc.), and downloadable applications. The Mobile Application System 105 communicates with the carrier services 104, for example, across a high bandwidth communications channel 108 or a publicly accessible network, such as the Internet 107, to provide provisioned applications to the subscriber devices 101. One skilled in the art will recognize that the Mobile Application System 105 may be fully or partially integrated with the carrier services 104. Using walled-garden provisioning, downloadable applications, which are typically generated by the content providers 106, are supplied to the Mobile Application System 105 or to the carrier services 104 directly or across a network, such as the Internet 107. These downloadable applications are then verified and customized as appropriate by the Mobile Application System 105 and provisioned for a subscriber device

101. In an embodiment that supports open provisioning, a subscriber of the carrier may download an application from a website by specifying a location (such as a network address, or URL – Uniform Resource Locator). The MAS intercepts the download request from the subscriber and then locates, verifies, and provisions the application for the customer.

The subscriber device 101 relies on a client-side application management utility (*e.g.*, a Handset Administration Console or a browser) to request and download applications. Figure 2 is an example block diagram of a Handset Administration Console that operates with a Mobile Application System. The Handset Administration Console handles notification, installation, and uninstallation of applications on the subscriber's wireless device. The subscriber device 201 provides the subscriber with a menu of functionality available for the device. The subscriber may select from the menu routines that, for example, manage applications already installed on the device and present new applications that can be downloaded. For example, the routines may allow the subscriber to obtain version information for installed applications, to download updates for those applications when they become available, and to browse for new applications to be downloaded. Menu 202 is an example menu showing a list of new applications 203 that can be potentially downloaded to the subscriber device 201. Example screen display 204 shows an example user interface that is presented after the subscriber has selected an application for downloading. Screen display 204 presents an icon 205 that depicts the downloading operation, the title of the application being downloaded 206, and a status bar 207 that displays the ongoing process of the download. The screen 204 also presents a stop button 208, which allows the downloading process to be canceled.

Figure 3 is an example overview flow diagram of the general steps performed by an example Mobile Application System to provision applications for wireless subscriber devices. These steps are applicable to either provisioning scenario – using walled-garden or open provisioning. These same steps can also be used to provision applications for wired devices, such as those connected through the Internet 107 in Figure 1. Steps 301-310 demonstrate how the MAS handles an incoming request to download an application from a subscriber device, provisions the requested application, and sends the requested application to the subscriber device. Provisioning includes one or more of the steps of retrieving, inspecting, optimizing, instrumenting code, and packaging, and may include additional steps as needed to ready an application for downloading to a target device. For example, as additional security and billing methods are added to the system, provisioning may include steps for encrypting and reported information. Where distinct, steps 301-310 assume that an application is being requested directly from the MAS as opposed to indirectly by browsing a location on a network. (In the case of open provisioning, the MAS intercepts the request and attempts to provision and download the application as though receiving it for the first time.)

Specifically, in step 301, applications are made available for downloading, typically from a carrier or directly from a content provider. Applications may be written using a computer language, such as Java, which is capable of executing on a wide variety of subscriber devices. The applications are stored locally in a carrier's application data repository (which may be located in the MAS or at the carrier's premises) or are optionally stored in trusted third-party servers. (In the case of open provisioning, the third-party servers are not necessarily trusted.) In step 302, the subscriber sends a request to download an application, retrieve a list of available applications, perform some administrative query, or other command. Protocol conversions are performed on incoming requests (and outgoing messages) to enable communications with subscriber devices across a wide variety of wireless carriers. The downloaded application may be, for example, a new and popular application or an upgrade or a more recent version of software that will run on the subscriber device. Requests are made, for example, using Uniform Resource Locators (URLs) that use the HTTP messaging to direct the requests. The MAS supports an extensible command processing engine and supports the direct invocation of the various handlers, modules, and other structures that are components of the MAS, either through HTTP requests or through an application programming interface (an "API"). In the case of an application provisioning request, the request to download a particular file is made by designating a URL that identifies a file (an application or service) to download. In the case of an administrative query, a request is made to, for example, an administrative servlet or other code in the MAS, which handles the request. In step 303, the MAS determines whether the request is for downloading or for some other command, and, if so, continues in step 305, else processes the command in step 304. In step 305, the MAS determines whether the designated URL specifies a published application (thereby indicating that walled-garden provisioning is to be performed), and, if so, continues in step 306, else continues in step 308. In step 306, the subscriber's request is

verified for authorization, device capability, and if appropriate, pre-paid billing authorization. The authorization level will typically depend on the level of service to which the client has subscribed. For example, in one embodiment the MAS supports pre-paid billing, which allows a subscriber to pay ahead for use of applications. In this case, the MAS will verify that the pre-paid billing account can cover the request before the application is downloaded. Other factors may apply such as whether a promotional offer is being tendered, the number of times the subscriber has accessed the service, whether an introductory offer exists, the time of day or week that the service is being accessed, the number of bytes to be downloaded, and other such factors. The device capability is also examined to determine whether the requested application can be run satisfactorily on the subscriber device. This can be performed, for example, by comparing the requesting device to known device profiles and an application profile for the requested application. In step 307, the MAS determines whether the subscriber's request has been successfully verified and, if so, continues in step 308, else it declines the request and returns to step 302 to await another request.

In step 308, the MAS determines if a pre-provisioned application already exists that corresponds to the subscriber request and is suitable for the subscriber device. A pre-provisioned application is an application that has been pre-customized according to the level of authorization and the capability of the subscriber device. Pre-provisioned applications, when available, minimize system latency and enhance system response time for a corresponding application request. Applications may be pre-provisioned according to typical levels of subscription of subscribers and typical subscriber devices (as determined, for example, by projected use) and stored for later access to respond to a subscriber device request for an application that corresponds to a pre-provisioned application. If the application has not been pre-provisioned, the MAS provisions the application dynamically, which will increase the time required to process the request, but will produce a customized and authorized application for deployment.

In step 308, if a suitable pre-provisioned application has been found for the subscriber device, the provisioning scenario continues in step 310, else it continues in step 309. In 309, the application is provisioned for the specific subscriber device and according to access authorization. In step 310, the MAS sends off the provisioned application to the subscriber device for downloading.

As mentioned, one of the requests supported by the MAS is to retrieve a list of available applications that can be downloaded to the subscriber's device. This process is referred to as application discovery. Figure 4 is an example overview flow diagram of the steps performed by an example Mobile Application System to perform application discovery for wireless subscriber devices. In an example embodiment, two types of application discovery are supported. The first is driven by the system and generates a system-derived list. The second is driven by the requester and specifies search terms that are matched by the MAS to generate a list of "suitable" applications. In step 401, the MAS determines whether the user has supplied any search terms, and, if so, continues in step 402, else continues in step 403. In step 402, the MAS searches a data repository of published applications for those that meet criteria specified in the request, and continues in step 404. Alternatively, in step 403, the MAS determines an initial list. In one embodiment this list is formed from a subscriber's personalized list if one is available, otherwise the MAS supplies a default list. In step 404, the MAS filters this initial list based upon subscriber and device capabilities. For example, the MAS may analyze various profiles, for example a subscriber profile, a device profile, and an application profile to determine whether the subscriber is authorized to use the application and whether the application's needs, as reflected in the application profile, are met by the device, as reflected in the device profile. In step 405, the MAS adds any system defined applications to the list (referred to as the "startdeck"). Such applications may be designated according to customizable rules of the carrier, for example, applications that generate more revenue may be given "premium" viewing time by placing them at the top of the list. In step 405, the MAS formats the list according to the viewing capabilities of the requesting device (for example, the markup language supported), and ends processing.

Figure 5 is an overview block diagram of the components of an example embodiment of a Mobile Application System. In this embodiment, the Mobile Application System 500 includes a Protocol Manager 503, Provisioning Manager 504, Cache 505, Deployment Manager 506, Billing Manager 507, Logging Manager 508, Administrator 509, and Heartbeat Monitor 310. These components inter-operate to receive applications from content providers and carrier services, to provision them for delivery to the subscriber devices, such as those shown in Figure 1, and to process MAS commands. One skilled in the art will recognize that many different arrangements and divisions of functionality of the

components or different components of the MAS are possible. For example, the functions allocated to the Protocol Manager 504 and the Billing Manager 507 could be combined in one component. Other arrangements are also possible and contemplated.

The various components of the MAS inter-operate to provide a multitude of capabilities to carrier (or system) administrators or customer care representatives who administer the services provided by the carrier, content providers who develop and distribute applications and services to the carriers, and subscribers who consume the services, applications, and other content. The Administrator 509 provides various user interfaces to each of these types of users to configure the MAS, applications, billing and other services, and to customize a subscriber's experience with the MAS. To illustrate the provisioning aspects, the functionality of the MAS is described from the point of view of the processing steps that occur in the MAS components when a subscriber invokes the MAS to download applications to the subscriber's device, as described with reference to Figure 3. One skilled in the art will recognize that other data flows and usages of the components are appropriate and depend upon the commands processed and/or how the components, or code within them, are invoked.

More specifically, in the example embodiment shown in Figure 5, communications from subscriber devices, such as J2ME or WAP handsets, are presented to and received from the Mobile Application System 500 as Incoming Request 501 and as Outgoing Data 502, respectively. Typically, the MAS is invoked by a subscriber via the command interface (as opposed to the website based interfaced) to process two different types of input requests: discovery of applications and downloading of a requested application. The MAS may also be invoked to process other commands. Also, components of the MAS, for, may be invoked directly, such as to perform administrative requests to obtain usage information. When input request 501 is a request for application discovery, the MAS compiles and returns a list of applications that are available and appropriate based on the subscriber, application profiles, and device profiles. The steps typically executed by the MAS to accomplish application discovery are described with reference to Figure 4. Alternatively, when input request 501 is a request to download a designated application, the MAS retrieves the application, verifies that it is appropriate and permitted for download to that device and user, provisions and packages the requested application, and sends the packaged application to the requesting subscriber device. The steps typically executed by the MAS to accomplish provisioning applications were described with reference to Figure 3.

The Protocol Manager 503 performs protocol conversion of the messages between the subscriber devices and the Provisioning Manager 504. Protocol conversion ensures that the MAS 500 can communicate with any subscriber device (wired or wireless), independent of the communication protocol used in the network (such as wireless network 102 in Figure 1), and allows incoming requests that may be embedded within various protocols to be processed. An example Protocol Manager 503 has built-in support for WAP and HTTP protocols and can be extended using well-known techniques to provide support for additional formats and protocols. One or more separate gateways such as a WAP gateway (not shown), may reside between the Protocol Manager 503, the incoming request 501/outgoing data 502. These gateways may be used to process messages targeted for a particular protocol. The Protocol Manager 503 may also optionally include a plug-in security layer to handle data encryption and decryption as well as certificate management for end-to-end security support. One skilled in the art will recognize that the Protocol Manager 503 can be extended to include other types of support for secure communications as desired.

After the incoming request is appropriately converted, the Provisioning Manager 504 processes the request, engaging the assistance of other components as needed. For example, if the request is an administrative query, then the Provisioning Manager 504 may forward the request to an administrative servlet in the MAS. If, instead, the request is for a list of applications that can be downloaded to a subscriber's device, then the Provisioning Manager 504 may interrogate the Data Repositories 311 and profile management code to generate such a list by comparing the capabilities and requirements of each application available from the carrier with the appropriate device and subscriber profiles that correspond to the subscriber's device and the subscriber. If, on the other hand, the request is from a subscriber to download a designated application, then the Provisioning Manager 504 and Deployment Manager 506 interact to obtain and ready the requested application for distribution to the subscriber. In one embodiment, the Provisioning Manager 504 verifies the user, device, billing, and application information referred to by a subscriber request and the Deployment Manger 506 retrieves and provisions the applications. The application provisioning process performed by the Deployment Manager 506 comprises one

or more of the following processing steps: retrieving, inspecting, optimizing, instrumenting code, and packaging, which are discussed below with reference to Figure 7.

The Provisioning Manager 504 receives subscriber requests from the Protocol Manager 503 and handles download requests or other commands that are contained in the subscriber requests. The download requests are handled based on information submitted with each download request and other information that is accessible by the MAS (for example, profiles store in data repository 511). When processing a request to download an application, the Provisioning Manager 504 examines previously created or available profiles for the subscriber, the subscriber devices, and the requested application(s) and information related to billing to determine the suitability of the requested application for the subscriber using the particular subscriber device and according to the subscriber's billing method. After inspecting the profiles, the Provisioning Manager 504 either approves or denies the request by attempting to evaluate, for example, whether the requested application can be successfully run on the subscriber device. This evaluation is performed, for example, by determining whether the requirements of the application can be met by the capabilities of the particular subscriber device. The Provisioning Manager 504 also determines whether the billing method that has been set up for the requested application and the subscriber is compatible and sufficient to perform the download. For example, if the request indicates that the subscriber is part of a pre-paid billing program, then the Provisioning Manager 504 verifies that the subscriber's pre-paid billing account funds are sufficient to allow the application download.

Once approved, the Provisioning Manager 504 may obtain the requested application from either the cache 505 or from the Deployment Manager 506. Typically, the cache 505 is used to store frequently downloaded applications in a pre-provisioned format, while the Deployment Manager 506 is used to provision applications dynamically, as they are requested. Applications that are controlled by the carrier are typically pre-provisioned and stored in the cache 505, while applications available through, for example, an Internet site, are typically provisioned only when requested for download.

The cache 505 is used to provide faster delivery of the requested application to the subscriber device. The cache 505 is used to cache provisioned applications that have been processed ahead of time for specific profiles such as for specific subscriber devices or according to authorized access. Applications stored in the cache 505 that have already been inspected, optimized, and instrumented are tagged as being ready for deployment. One skilled in the art will recognize that system performance may be enhanced by implementing similar caching functionality between other components of the MAS as well. For example, a cache to hold Internet applications, which resides between the Deployment Manager and the Internet, could reduce the access time required for communicating with Internet applications. Also, for example, a cache to hold unarchived JAR files could be implemented to speed up the instrumentation process. Other configurations are also possible. If an approved requested application for a particular subscriber and particular device is not found in the cache 505, it can be retrieved via the Deployment Manager 506. The Deployment Manager 506 prepares applications for delivery to a subscriber device. The Deployment Manager 506 manages many facets of preparing, maintaining, and provisioning applications, such as malicious application detection, restricted API usage, support for trial distribution (use allowed for only a set number of times or a set period of time) and other billing methods, application size optimization for the requesting subscriber devices, and other facets. The Deployment Manager 506 obtains applications and provisions each application instance for its intended (requested) use when an instance of an application is requested. It may also pre-deploy ("pre-provision") applications for specific device and/or subscriber profiles by preparing applications for those profiles in advance and storing the results for quick access in the cache 505, or other data repository. As is discussed below with reference to Figure 7, the Deployment Manager 506 may deploy applications from a carrier's application data repository or from remote application hosts (trusted or otherwise), or from any other application source. After the Deployment Manager 506 has suitably provisioned the requested application, it sends the requested application back to the Provisioning Manager 504 for any postprocessing to the outbound response.

As a provisioned application is being delivered to a user, the details about the transaction typically are recorded in the Logging Manager 508, which is accessible to the Billing Manager 507 to enable a variety of billing methods. The recorded data includes information pertaining to the incoming request 501 and the deployed application such as the subscriber ID, the size of the download, the time and date of the download, the particular application downloaded, etc. Because of the wide range of information recorded about the download, the carrier has great flexibility in methods of billing for the provisioning of

applications according to different categories of service and subscribers. The carriers can bill, for example, by the amount of airtime used, the time of download, the amount of data downloaded, the demographics of the client, or on the basis of the particular application that was downloaded.

The Billing Manager 507 is responsible for assisting in the enforcement of billing methods. In an example embodiment, several initial billing options are provided: (1) download charges based upon downloading an application; (2) packet-based billing charges based upon transmissions of network packets; (3) subscription charges based upon periodic fees such as daily, weekly, or monthly; (4) trial use charges based upon any metric of trial use, for example the number of times an application can be executed; and (5) pre-paid billing. These billing options are customizable at both the carrier level and the application level, and, when more than one is offered for a particular application, a desired billing option may be selected by a subscriber. In an example Mobile Application System 500, an application programming interface (API) is provided for easy integration with a carrier's existing billing subsystem. If a carrier supports pre-paid billing, a subscriber can establish an account that is maintained by the carrier. In one embodiment, the subscriber prepays for applications to be downloaded at a later time. When the subscriber downloads a pre-paid application, the Billing Manager 507 forwards a billing record to the pre-paid billing system of the carrier so that the subscriber's account can be charged and updated. In an alternate embodiment, pre-paid subscriber accounts are stored and maintained by the Billing Manager 507. Other configurations are also possible, as well as support for other types of billing methods. After the Billing Manager 507 has generated billing related information, the application is forwarded to the Protocol Manager 503, where it is then reformatted for a different protocol if required and transmitted to the customer as outgoing data 502.

The Administrator 509 interacts with the other components of the example MAS 500 to customize various aspects of the MAS 500. For example, the Administrator 509 allows carriers to implement customizable provisioning-related policies and integrate the MAS with their own infrastructures through reprogramming components of the Mobile Application System itself, thereby allowing subscribers, carriers, system administrators, and content providers enhanced flexibility in performing profile management, report generation, billing method administration and server administration.

The Heartbeat Monitor 510 monitors and provides reports on other MAS 500 components and provides appropriate notifications when relevant system events occur, for example, to detect problems in the system such as a component becoming inoperative. For example, the Heartbeat Monitor 510 can monitor the Protocol Manager 503 to determine if the Protocol Manager 503 responds to an incoming request within a predetermined time limit. If the Heartbeat Monitor determines that the Protocol Manager 503 is not properly responding, it can flag the event and notify a system administrator. In one embodiment, multiple Heartbeat Monitors 510 are provided so that a second monitor can monitor whether the first monitor is functioning properly and take over if necessary. The Heartbeat Monitor 510 is capable of both active monitoring (by polling devices with status requests) and passive listening (by verifying that specific types of communications occur at appropriate times). The Heartbeat Monitor 510 also provides interfaces to industry standard protocols, for example Simple Network Management Protocol (SNMP), to enable other external code to monitor the MAS.

As described with reference to Figure 5, the Provisioning Manager of the MAS processing the incoming download requests and other commands, and drives dynamic provisioning of applications for downloading. Figure 6 is an example block diagram of the components of an example Provisioning Manager of a Mobile Applications System. In one embodiment, the Provisioning Manager 600 comprises a MAS Command and Control Processor 620 (the "MCCP"), Verifiers 601, XSLT processor 630, Request Preprocessor and Postprocessor 640, and MAS Data Query Engine 650. The MCCP is responsible for decoding the request and directing it to the correct MAS subcomponent, for example, to download a published application or to perform application discovery. The Verifiers 601, which comprise Subscriber Verifier 602, Device Verifier 603, Pre-Paid Billing Verifier 604, and Application Verifier 605, perform verifications to determine suitably of an application for a subscriber and a device. The XSLT processor (which can be implemented, for example, as an industry standard Extended Stylesheet Processor) is used to format the data according to the rendering capabilities of the requesting device. In one embodiment, it supports stylesheets for XML, but can easily be extended to provide additional stylesheets for HTML, Java, WML, XHTML Basic, and text, or any other markup or rendering language. The Request Preprocessor and Postprocessor 640 manipulates parameters in the request "packets"

to communicate amongst the other components, and can be extended to perform any type of processing that can be "hooked" in at this level. The MAS Data Query Engine 650 manages communication with the various data repositories. It includes readers for Provisioning 651, Profiles 652, and Configuration Data 653. Although no arrows are shown connecting these components for ease of viewing, one skilled in the art will recognize that the components are interconnected and inter-operate in many ways.

Initially, the Provisioning Manager 600 receives an incoming request such as from the Protocol Manager (for example, Protocol Manager 504 of Figure 5). The Provisioning Manager 600 optionally preprocesses the request by analyzing the incoming request and modifying the request dynamically to allow for enhancing, altering, or limiting the provisioning, billing, or logging steps to be taken later. Such dynamic modification enables carriers to hook their own infrastructure dynamically into the system. For example, the Provisioning Manager 600 can look at the request headers passed along with the incoming download request and modify, add, or remove the headers to modify the behavior of the system. Because other components in the MAS use information contained with the headers to perform their functions, updating or modifying the header information provides a means to extend or limit the functionality of a specific request, or modify the behavior of the MAS.

The request, when received from the MAS command interface (as opposed to directly invoked via website or API) is processed by the MCCP. If the request is for application discovery or to download content, various Verifiers 601 are used to determine compatibility of an application. If the request is for some other command, then it is processed accordingly.

The Application Verifier 604 determines whether a requested application has been forbidden by the carrier for deployment. Specifically, the Application Verifier 604 examines a list of applications that the carrier does not want to allow to be downloaded to determine if the carrier has banned the requested application. This situation could occur, for example, if an application has been suddenly found to provide malicious behavior and the carrier wants to immediately halt its distribution.

The Subscriber Verifier 601 determines the identity of the subscriber from whom the request originated and determines the level of services to which the subscriber is entitled to determine whether the subscriber is authorized to use a specific application. The particular services to which the subscriber is entitled may be determined by retrieving, using the Profile Reader 652, a corresponding subscriber profile and examining a variety of factors, either singly or in combination. Factors may, for example, include the number of downloads permitted within any month, the time required for downloads, the time of day and time of week in which the request is made, the availability of special offers and grace periods, etc. The Subscriber Verifier 601 also can determine a subscriber group to which a subscriber belongs and determine the level of access permitted to the subscriber by determining the services that are allowed and not allowed for the subscriber group as a whole. An example embodiment of the determination performed by the Subscriber Verifier is described with reference to Figure 17.

The Device Verifier 602 determines the type and capabilities of the subscriber device from which the request was made and determines whether the device capabilities are sufficient to support a specific application. The capabilities of the subscriber device are determined by retrieving using the Profile Reader 652 a device profile, if one exists, that corresponds to the requesting subscriber device. The device profile is examined to determine whether the device has the characteristics required by the requested application to execute properly on the subscriber device. An example embodiment of the determination performed by the Device Verifier 502 is described with reference to Figure 18.

When a pre-paid billing method is supported by the MAS, the Pre-Paid Billing Verifier 603 queries the carrier pre-paid billing infrastructure, wherever billing records for individual subscribers are stored. A download request is allowed to proceed to provisioning, typically only if there are sufficient funds in the subscriber's account, as indicated by the carrier.

After the Provisioning Manager 600 has determined that the subscriber device is suitable to run the requested application, the subscriber is authorized to use the application and has sufficient funds (if part of a pre-paid billing scheme), then the Provisioning Manager 600 invokes a provisioning interface of the Deployment Manager to obtain a corresponding provisioned application. The Deployment Manager, which is described with reference to Figure 7, retrieves and provisions the requested application and returns it to the Provisioning Manager 600.

After a provisioned application suitable for downloading to the subscriber device is obtained from the Deployment Manager, the Provisioning Manager 600 optionally postprocesses the request. As with preprocessing, postprocessing may perform additional modifications to the verified request so that the modifications can be used to extend the functionality of the MAS. For example, instructions can be associated with the request that will later direct the Protocol Manager (for example, Protocol Manager 503 of Figure 5) to package the request for a custom protocol.

As mentioned, the Deployment Manager (such as Deployment Manager 506 of Figure 5) receives a subscriber request from the Provisioning Manager or receives a direct request (such as from a system administrator) to obtain a provisioned application that corresponds to the request. The request includes a URL of the requested application, which indicates a source location for the application. In one embodiment, the URL references a list of mirror sites and retrieves the application from an optimal location that is determined from the MAS. In another embodiment, the URL is a proxy and the Deployment Manager redirects the URL to its actual location. Such methods can provide additional security layers to the system. One skilled in the art will recognize that any method of indicating a location for the application can be used with these techniques, and that these techniques operate on indicators other than URLs. The application is also inspected, optimized, and instrumented for delivery before it is deployed and sent to the subscriber.

Figure 7 is an example block diagram of the components of a Deployment Manager of a Mobile Application System. The Deployment Manager 700 comprises a Retriever 701, Remote Fetcher 702, Local Fetcher 703, Inspector 704, Optimizer 705, Instrumentation Installer 706, and Application Packager 707. The Retriever 701 is obtains the application code from the proper host server using either the Remote Fetcher 702 or the Local Fetcher 703 and then passes the application code through a variety of components to properly provision the application code. In particular, the Inspector Component 704 inspects the application for malicious code and forbidden API; the Optimizer Component 705 reduces the size of the code if possible; and the Instrumentation Installer 706 incorporates carrier specified policies and administrative features, for example billing and notification messages, into the code.

Specifically, the Retriever 701 is designed to allow multiple users and multiple carriers to communicate over a variety of networks using different protocols. This is accomplished, in part, by allowing carriers flexibility in the locations of the software applications (content) that they host for distribution. For example, carriers may choose to host all available applications from their own network by storing such applications in designated directories on an FTP or HTTP server or data repository, such as a standard DBMS. The Carrier Application Store 708 is such a data repository, and may reside on a server of the MAS itself. The Retriever 701 activates Local Fetcher 703 to retrieve a copy of the locally stored data. Carriers may also choose to allow trusted third-party application providers to host the applications from Remote Application Hosts 709, which are under the control of the trusted third-party application providers. In addition, when used to perform open provisioning, the Retriever 701 can retrieve applications from third party hosts that are not necessarily from trusted sources. In both cases, the carrier uses a URL supplied by the third party to refer the incoming request to a particular downloadable application that is hosted on one of the Remote Application Hosts 709. The Retriever 701 typically activates the Remote Fetcher 702 to retrieve such applications hosted on Remote Application Hosts 709, when such hosts are accessible via public protocols. In one embodiment, the Local Fetcher 703 may be optimized to quickly retrieve locally stored data, whereas the Remote Fetcher 702 implements the public protocols necessary to retrieve applications that reside on hosts that are accessible across a public network. .

Depending upon preferences of a trusted third party host or the carrier, the application code retrieved by the Retriever 701 may be already provisioned. If the Retriever 701 obtains unprovisioned code, the code is sent to the Inspector 704, Optimizer 705, and Instrumentation Installer 706 for further processing. The Inspector 704 examines the retrieved unprovisioned application code to detect malicious code. On Java code, the Inspector 704 may also perform a class analysis of the application code to verify that classes in the application conform to desired standards such as the number, type, and frequency of API calls. In addition, the Inspector 704 applies application filters to detect package and method names, classes, fields or other forms of an API that are suspected to have intrusive, malicious behavior, or that may be unauthorized for use by the requesting subscriber, the target device, or some other target. The Inspector 704 may also apply application filters to

detect API usage patterns. The Inspector 704 has available for its use the subscriber and device profiles that were retrieved by the Provisioning Manager (as described with reference to Figure 6) so that the Inspector 704 can enforce restrictions on a per device or per subscriber basis. In an example embodiment, the Inspector 704 allows the thresholds of such parameters to be adjusted, as well as the thresholds for flagging parameters for further inspection by other entities such as the Logging Manager, for example. If the Inspector 704 discovers potentially malicious behavior, the provisioning (and subsequent downloading) may be prevented (or the subscriber warned), and the violation along with the identity of the offender reported to the Logging Manager. .

After the Inspector 704 has successfully examined the retrieved unprovisioned application code, the code is forwarded to the Optimizer 705 for further processing to reduce the size of the application. The Optimizer 705 uses well-known methods in the art to shorten variable names and to remove unused code from the application. Such optimization procedures typically result in faster downloads. The Optimizer 705 may also use techniques that are well-known in the art to increase the speed of the application when it is executed, such as changing the use of particular instructions to more efficient instructions. One skilled in the art will recognize that, because components of the MAS may be extended or modified, any optimization technique can be incorporated into the system.

After optimization, the inspected, optimized application code is forwarded to the Instrumentation Installer 706 for further processing. Because the suppliers of downloadable applications typically do not have the ability to modify the requested applications for individual subscribers, it may be desirable to modify the code of an application to add subscriber-specific code. For example, billing options such as a "trial use" scheme can be implemented by inserting code into the application that causes, for example, the application to only execute a certain number of times or for only a specified period of time. Similarly, code that reports information for logging purposes or code that collects information for other billing options (such as packet-based billing which charges based upon the number of network packets transmitted) can be instrumented. Also, in the case of open provisioning, code that warns the subscriber that the subscriber is about to download and execute content from an untrusted source can be instrumented. The Instrumentation Installer 706 can also modify the code in the application according to other policies that are specified by carriers, for example, policies that implement promotions and advertising campaigns. One skilled in the art will recognize that code can be instrumented for many other purposes as well can be instrumented in predetermined locations using well-known methods such as manipulating libraries or by subclassing classes and methods.

After the Instrumentation Installer 706 has instrumented the requested application, the Application Packager 707 packages the inspected, optimized, and instrumented application. The Application Packager 707 packages the requested application by formatting the contents of the application file in a manner that the subscriber device can read, as determined from the device profile that was obtained by the Provisioning Manager, as described with reference to Figure 6. For example, many subscriber devices are capable of reading files that are presented in a compressed "JAR" format (a Java archive format), which is a format used to compress and package requested Java applications. Because some devices may not accept a compressed JAR file, the Application Packager 707 provides custom packaging of provisioned applications for those subscriber devices that cannot accept files in compressed JAR format. One skilled in the art will recognize that such packaging converters and other converters for formats other than JAR can be installed into the Application Packager 707 using well-known techniques, such as by subclassing the packaging routines. In addition, some subscriber devices may limit the size of packets that they can receive. When detected, the Application Packager 707 can package a provisioned application for such a subscriber device into multiple data files that the subscriber device can assemble into a single JAR file upon receipt, which can then be used by the subscriber device to install the application.

As mentioned with respect to Figure 5, the Administrator component (*e.g.*, Administrator 509) may be used by different types of users to configure the various components of the Mobile Application System and to specify preferences. Figure 8 is an example block diagram of an Administrator component of a Mobile Application System. In one embodiment, the Administrator 800 preferably provides multiple Web-based user interfaces to allow content providers, system (carrier or MAS) administrators, subscribers, and customer service support staff to access the components of the MAS or to customize their experiences. In particular, the example Administrator provides a Content Provider Website 801, an Administration Website 802, and a Personalization Website 803. One skilled in the

art will recognize that each described website may comprise multiple screen displays, and that these and/or other screen displays and websites may be combined in various configurations to achieve the same result. For example, the Administrator Website 802 may be optionally include a separate Customer Care Website 804, which can be used by a customer care representative (of typically the carrier) to manage individual subscriber accounts on behalf of the carrier.

The Administrator 800 provides a Content Provider Website 801 for content providers to use to submit downloadable applications to the MAS and to monitor whether the submitted downloadable applications have been reviewed (*e.g.*, inspected) and approved for publication. Content providers can also use the Content Provider Website 801 to recommend changes to an application profile, to monitor the popularity of their applications, or to send communications to a MAS administrator.

The Administrator 800 also provides an Administration Website 802 for MAS system administration, for example, to manage the published and pending applications submitted by content providers. In one example embodiment, the Administration Website 802 interface provides separate nodes to establish, configure and/or manage accounts, applications, subscribers, devices, servers, and reports.

The system administrator can use an application node of the Administration Website 802 to specify global billing methods supported by the carrier. In one embodiment, the administrator can select multiple different billing options for charging per download, per network usage (*e.g.*, transmission- based), and per subscription, and trial use free of charge.

Other functions are also accessible to system administrators via the Administration Website 802. For example, system administrators may use the subscribers node to manage the use of the MAS by subscribers and to establish a subscriber profile for each subscriber. The subscriber profiles maintain lists of published applications that have been downloaded by each subscriber, maintain a list of banned applications that a particular subscriber may not run, and creates and maintains the subscriber groups to which the particular user belongs. In one embodiment, these profiles are stored in a data repository in the MAS (such as data repository 511 in Figure 5) and read by the Profile Reader of the Provisioning Manager (such as Profile Reader 652 of Figure 6).

The system administrator may also send a subscriber a message, such as a notification that an updated version is available for one of the applications already downloaded by the subscriber. This behavior is sometimes referred to as "push" capability. Information for contacting the subscriber is available typically from the subscriber's subscriber profile.)

In addition, system administrators can choose to remotely activate or deactivate downloaded applications over the wireless network provided the Instrumentation Installer 706 has appropriately instrumented the downloaded content. For example, instrumented applications can be forced to check with the host server (carrier or third party) to see if a new version of the application is available and can prompt the subscriber to determine whether to download the new version of the application. Instrumented applications also can be forced to check with the host server to determine if a time limit period has expired or the number of times the application can be run has been exceeded (for example, for use with a trial period billing option). Instrumented applications may also place time of day restrictions that may, for example, restrict an application to be used only a certain number of times within a set time period of a day. These restrictions effectively allow system administrators to revoke or restrict the privilege of a subscriber to execute an application even after the application has been downloaded to the subscriber's wireless device. One skilled in the art will recognize that other restrictions and capabilities may be similarly enforced.

System administrators can use the devices node of the Administration Website 802 to submit and maintain information that is used for verification during the provisioning of an application. For example, system administrators can create and maintain a list of device profiles that correspond to particular devices. Typically, the system administrator creates a device profile for each device that is supported by the MAS. New device profiles and corresponding device designations may be added as necessary. Each device profile contains hardware specific information and resource characteristics, such as the amount of runtime memory and flash memory, chip identification, maximum download size, and whether the device is "OTA" compliant. (OTA refers to Sun Microsystem's Over The Air conformance specification. Devices that conform to OTA support the tracking of successful downloads on devices amongst other capabilities.)

Each device profile can also designate a single Java profile that is supported by the device. A Java profile specifies the Java API that is supported by a device. For example, a device that conforms to the MIDP 1.0 standard (a well-known standard that defines a set of Java API implemented by the device) would typically have a device profile that indicates this conformance. The device (and associated Java) profiles are used by the Provisioning manager during the verification process for comparison with an application profile to insure that a particular device has the resources and supports the set of API required by the application. Although the same Java profile can be associated with multiple devices, a device can typically only support one Java profile. A system administrator loads a Java profile by specifying the name of an archive file (*e.g.*, a JAR file or a .zip file) that specifies and describes a set of APIs. The MAS examines the specified archive for its structure (package, class, method, and field definitions) and generates a profile that contains this structure. These profiles can be also used to create application filters, which prevent the provisioning and/or downloading of applications that use specific API. Although discussed primarily with respect to Java-enabled devices and Java API, one skilled in the art will recognize that the MAS can be adapted to other language specifications and other language enabled devices, as long as the language supports a determination of whether particular API, objects, classes, variables, and/or other data structures are present in an application and supported by devices and as long as structure can be ascertained at the byte-code level. In addition, one skilled in the art will recognize that these techniques can be adapted for use at the source code level, as long as the receiving application can compile or interpret the source to generate an executable file.

The Administration Website 802 enables system administrators to implement various security techniques and policies that supplement and complement the verification and inspection processes provided by the Provisioning and Deployment Managers. One such technique is the ability to define application filters, as discussed, which are used to specify API that should not be called by an application using a particular device or other target. Such restricted calls and structures can be identified during the application provisioning process in response to a subscriber request to download and upon submission of an application by content providers to help ensure that a subscriber will not load code that is inappropriate for a particular device. Another security technique provided is the ability to redirect URLs.

System administrators can redirect URLs for the convenience and security of users of the MAS by specifying URL redirection mappings using the Server node of the Administration Website 802. For example, a URL that points to an unauthorized advertising site may be redirected to a URL that provides advertising from a paying advertiser. Similarly, after removing content, the system administrator may wish to redirect the URL that previously referred to the content instead to an error message. Also, redirected URLs may be used to hide the real location of an application or to enable an application to be moved more easily. Upon receiving incoming data, the MAS compares any URLs that specify an application with a list of redirected URLs managed using the Administration Website 802 and redirects them if so specified. One skilled in the art will also recognize that additional and other security techniques can be added to and utilized by the MAS and, where necessary, configured through the Administration Website 802 to provide a variety of security mechanisms to securely communicate between subscribers, content providers, administrators, and various MAS components and to securely transport data stored in the MAS, accessible through the MAS, or stored on the client device. For example, as devices are manufactured that support secure protocols such as KSSL, various MAS components can be configured to use the protocols. In addition, where applicable, secure interfaces can be installed as components between the web-based interfaces and the MAS components manipulated by them.

The Administrator 800 also provides a Personalization Website 803, which is used by subscribers to order, maintain, and display services and information related to the subscriber and to manage applications. A subscriber uses the Personalization Website 803 to subscribe to additional categories of content by changing service plans (which may possibly cause a change in the amount billed to the user). When a new service plan is selected, the subscriber is then authorized to use the associated content categories.

The subscriber may also manage the subscriber's applications by viewing current applications, adding applications, removing applications, and organizing applications. Although described with reference to applications, one skilled in the art will recognize that these techniques can be applied to any downloadable content and that the applications can be managed by category or other abstractions in addition to application-based management. A subscriber can use the Personalization Website 803 to create and manage the subscriber's "Personal Access List" ("PAL"). A subscriber's PAL is the list of applications that the

subscriber desires to have the MAS display on the subscriber's device during, for example, application discovery. This list may include a default set of applications, no applications, or a list of applications the subscriber has downloaded or desires to download at some future time, or other combinations. In one embodiment, the PAL initially contains what the subscriber has downloaded. Because the MAS maintains records of application downloads for each and every wireless device, the MAS is able to track a particular subscriber's downloaded applications.

A subscriber can also use the Personalization Website 803 to obtain and change account information and a history of download or account activities.

Through the Personalization Website 803, system administrators can notify subscribers of the availability of updated or new applications, or "tie-ins," by which system administrators can display product offerings or advertisements (through "push" messaging). A subscriber may access the Personalization Website 803 using the subscriber's wireless device or using a wired device that preferably has superior display characteristics over the wireless device (such as a personal computer). When a wired device having superior display characteristics is used to access the Personalization Website 803, superior display characteristics may be used to support enhanced tie-ins.

In addition to providing various website-based user interfaces to existing MAS components, the Administrator component of the MAS (*e.g.*, Administrator 509 in Figure 5) enables system administrators to implement customizable provisioning-related policies through reprogramming components of the MAS itself and through defining provisioning rules. In one embodiment, reprogramming is accomplished through the Administrator Website 802; however, one skilled in the art will recognize that this functionality can be accomplished using other mechanisms, for example, by registering different components and profiles with the Administrator through a registration mechanism, or by subclassing elements of the MAS interfaces. This functionality allows subscribers, carriers, and system administrators enhanced flexibility in reviewing submitted applications, performing profile management, report generation, and server administration.

For example, a system administrator can employ profile management to implement provisioning rules. Profiles provide a data-driven technique that is inherently dynamic. By specifying various categories of service for subscribers and groups of subscribers, provisioning rules may be applied to individuals or to groups of subscribers simply by modifying various profiles, for example using the website interfaces of the Administrator component. In addition, provisioning rules can be stored in profiles that are used to determine how the categories of service are applied to individual subscribers and to groups of subscribers. The provisioning rules themselves can be modified.

Profile management allows a high degree of flexibility in defining provisioning-related and billing-related service policies. For example, the carrier may offer subscription services comprising a basic service level and a premium service level. Subscribers of the basic service might be charged individually for each application they download, whereas subscribers of the premium service would pay higher a monthly service fee, but would be allowed to download an unlimited number of applications at no extra charge. In another example, an enterprise such as a bank could negotiate with the carrier to set up a specific type of service in which the enterprise's customers would be able to download an enterprise-specific application on one type of subscriber device to allow, for example, the bank's customers to look up account balances and transfer funds. In this example, the carrier hosts the subscriber profile for the enterprise and allows the enterprise to access this information using industry standard databases such as LDAP and relational databases that are well-known to one skilled in the art.

The Administrator 800 also provides the user interfaces necessary for administering other components of the MAS. Through these interfaces, system administrators can observe different modules of the MAS, manage server-side security, and monitor system status and server performance at any time. System administrators can also manage subscriber accounts and assign various levels of administrative privileges. Server administration also includes functions such as log management and analysis tools for troubleshooting purposes.

In example embodiments, the methods and systems of the Mobile Application System are implemented on one or more general purpose computer systems and wireless networks according to a typical client/server architecture and may be designed and/or configured to operate in a distributed environment. The example embodiments are designed to operate in a global network environment, such as one having a plurality of subscriber devices that communicate through one or more wireless networks with the MAS.

Figure 12 is an example block diagram of a general-purpose computer system and a subscriber device for practicing embodiments of the Mobile Application System. The computer environment of Figure 12 comprises a subscriber device 1201 and a general purpose computer system 1200, which communicate via a wireless carrier 1208. Each block may represent one or more such blocks as appropriate to a specific embodiment, and each may reside in separate physical locations.

The subscriber device 1201 comprises a computer memory ("memory") 1202, a display 1203, Input/Output Devices 1204, and a Central Processing Unit ("CPU") 1205. A Handset Administration Console 1206 is shown residing in memory 1202 with downloaded applications 1207. The Handset Administration Console 1206 preferably executes on CPU 1205 to execute applications 1207 currently existing in the memory 1202 or to download applications from the MAS 1209 via the wireless carrier 1208 as described with reference to the previous figures.

The general-purpose computer system 1200 may comprise one or more server and/or client computing systems and may span distributed locations. In one embodiment, the MAS is implemented using Java 2 Enterprise Edition (J2EE) and executes on a general-purpose computer system that provides a J2EE compliant application server. According this embodiment, the MAS is designed and coded using a J2EE multi-tier application architecture, which supports a web tier, business tier, and a database tier on the server side. Thus, general purpose computer system 1200 represents one or more servers capable of running one or more components and/or data repositories of the MAS.

As shown, general purpose computer system 1200 comprises a CPU 1213, a memory 1210, and optionally a display 1211 and Input/Output Devices 1212. The components of the MAS 1209 are shown residing in memory 1210, along with other data repositories 1220 and other programs 1230, and preferably execute on one or more CPUs 1213. In a typical embodiment, the MAS 1209 includes Provisioning Components 1214, Data Repositories 1215 for storing profiles and configuration data, and Applications Store 1216. As described earlier, the MAS may include other data repositories and components depending upon the needs of and integration with the carrier or other host systems. The Provisioning Components 1214 includes the components of the MAS illustrated in and described with reference to Figure 5. The Provisioning Components 1214 enable the MAS

1209 to receive requests for downloadable applications and application discovery, to verify the appropriateness of the request for use by a particular subscriber and a particular subscriber device, to customize the requested application accordingly, and to send the provisioned application to the subscriber device 1201. Applications Store 1216 is a data repository that stores applications suitable for downloading to the subscriber device 1201. The applications may be pre-provisioned ("static provisioning") for quick downloading to the subscriber device 1201, or the applications may be provisioned upon request ("dynamic provisioning"). The Data Repositories 1215 provide data repository and retrieval services to establish levels of subscription and device capabilities (to host the profiles used in profile management) and to determine applications suitable for each customer device.

One skilled in the art will recognize that the MAS 1209 may be implemented in a distributed environment that is comprised of multiple, even heterogeneous, computer systems and networks. For example, in one embodiment, the Provisioning Components 1214 and the Applications Store 1215 are located in physically different computer systems. In another embodiment, various components of the Provisioning Components 1214 are hosted on separate server machines and may be remotely located from the data repositories 1215 and 1216. Different configurations and locations of programs and data are contemplated for use with techniques of the present invention.

In an example embodiment, the Provisioning Components 1214 are implemented using a J2EE multi-tier application platform, as described in detail in *Java™ 2 Platform, Enterprise Edition Specification, Version 1.2*, Sun Microsystems, 1999, herein incorporated by reference in its entirety. The Provisioning Components 1214 include the Protocol Manager, the Provisioning manager, the Deployment Manager, the Billing Manager, among other components. Figures 13-28 describe various example embodiments of the specific routines implemented by each of these components to achieve the functionality described with reference to Figures 3-8. In example embodiments, these components may execute concurrently and asynchronously; thus, the components may communicate using well-known message passing techniques. One skilled in the art will recognize that equivalent synchronous embodiments are also supportable by a MAS implementation. Also, one skilled in the art will recognize that other steps could be implemented for each routine, and in different orders, and in different routines, yet still achieve the functions of the MAS.

Figure 13 is an example flow diagram of processing performed by a Protocol Manager of a Mobile Application System to communicate with various subscriber devices across varying wireless networks using different protocols. (*See*, for example, Protocol Manager 503 in Figure 5.) In step 1301, the Protocol Manager is initialized. In step 1302, the Protocol Manager determines whether there is an incoming data request from a subscriber device and, if so, proceeds to step 1303, else continues in step 1306. In step 1303, the Protocol Manager determines the protocol used for the incoming request by determining across which wireless network (or wired network) the request was sent, and stores the determined protocol for the pending request in a record associated with the incoming request. An association between the protocol record and the incoming request as it is processed by the system is maintained, for example, by storing a reference to the protocol record within the request message header. In step 1304, the Protocol Manager translates the incoming request to the internally used protocol (*e.g.*, HTTP). In step 1305, the Protocol Manager sends the translated request to the Provisioning Manager (for example, Provisioning Manager 504 in Figure 5), and then ends processing the request. In step 1306, the Protocol Manager determines whether or not there is a outgoing data request to a subscriber device and, if so, proceeds to step 1307, else ends processing the request. In step 1307, the Protocol Manager retrieves the determined protocol that is associated with the incoming request that corresponds to the output data. The determined protocol is the protocol used by the subscriber device that issued the request. In step 1308, the Protocol Manager encodes / translated the outgoing data message according to the determined protocol. In step 1309, the Protocol Manager transmits the encoded outgoing data to the subscriber device that submitted the request, and ends processing.

Figure 14 is an example flow diagram of processing performed by a Provisioning Manager of a Mobile Application System to determine the suitability of a requested application for a subscriber device and to present the requested application to the device in a format that the subscriber device can decode. (*See*, for example, Provisioning Manager 504 in Figure 5.) In step 1401, the Provisioning Manager performs any needed initialization. In steps 1402-1413, the Provisioning Manager processes a MAS "command." In step 1402, the Provisioning Manager determines the command (or request to download) that is specified in the incoming request. In step 1403, the Provisioning Manager preprocesses, as described with reference to Figure 6, the request by analyzing the incoming request and modifying the request dynamically to provide for enhancing, altering, or limiting certain provisioning steps to be taken later or by inserting other parameter values for communication or configuration reasons. In step 1404, the Provisioning Manager determines whether the "command" is a request to download and, if so, continues in step 1404, else continues in step 1408. Although currently implemented as separate "types" of commands, one skilled in the art will recognize that even though download requests are indicated by specifying a "URL" as a parameter, they are essentially commands and that there are many equivalent programmatic techniques for performing command processing. In step 1405, the Provisioning Manager determines whether the application (content) requested is known to the MAS and if so continues in step 1406 to perform walled-garden provisioning, else continues in step 1407 to perform open provisioning. In an example embodiment, there are several ways that content may become known to the MAS: through a system administrator using the website to provision and publish applications, through a content provider submitting content that eventually gets approved and published, and through a subscriber requesting the download of a yet to be known application from a trusted third party content provider known to the carrier which causes a submission process to indirectly occur. Walled-garden provisioning is discussed further with reference to Figure 15, and open provisioning is discussed further with reference to Figure 16. Once the provisioning has occurred in steps 1406 or 1407, the request is post-processed in step 1413. If, on the other hand, in step 1408 the designated command is a request for an application list, then the Provisioning Manager continues in step 1409 to perform application discovery, or continues in step 1410. After application discovery, the Provisioning Manager proceeds to step 1413 to post-process the request. In step 1409, if the command is a request for download history, then the Provisioning Manager continues in step 1411 to retrieve the list of downloaded applications, and proceeds to step 1413 for post-processing. In step 1412, if the command is some other MAS command, then the Provisioning Manager appropriately processes the command, and proceeds to step 1413. In step 1413, as described with reference to Figure 6, the Provisioning Manager post-processes the request by modifying the request to contain references to instructions for directing other components of the MAS to perform functions that are extensions of the other components' functionality or modifies other parameters. For example,

if the Provisioning Manager determines that the individual requesting the download is an employee of a highly valued advertising client, the Provisioning Manager may direct the Billing Manager not to bill for this particular transaction. After post-processing the request, the Provisioning Manager ends processing until another request is received.

Figure 15 is an example flow diagram of processing performed by a Perform Walled-Garden Provisioning routine of a Provisioning Manager. (*See* step 1406 of Figure 14.) In walled-garden provisioning, the requested application is verified for authorization by the subscriber and capability by the subscriber's device. Specifically, in step 1501, the Provisioning Manager retrieves the subscriber profile, the device profile, and the application profile that corresponds to the requested application. In one embodiment, these profiles are retrieved by invoking the Profile Reader 652 in Figure 6. In step 1502, the Provisioning Manager performs application verification to verify that the requested application has not been restricted by the wireless carrier, for example due to inclusion of forbidden APIs. Application verification is discussed further with reference to Figure 16. In step 1503, the Provisioning Manager performs subscriber verification to determine whether the subscriber is authorize via carrier billing policy or otherwise the requested application. Subscriber authorization is discussed further with reference to Figure 17. In step 1504, the Provisioning manager performs device verification to determine whether the device has the resources and other capabilities specified by the application profile that corresponds to the requested application. Device authorization is discussed further with reference to Figure 18. In step 1505, the Provisioning manager performs pre-paid billing verification, if pre-paid billing is included in the system, to verify that the subscriber's account is sufficient to be charged for downloading the application, as described with reference to Figure 6. In step 1506, the Provisioning Manager invokes the provisioning interface of the Deployment Manager, and returns the provisioned application.

Figure 16 is an example flow diagram of processing performed by a Verify Application routine of a Provisioning Manager. (*See* step 1502 in Figure 15.) In summary, the Verify Application routine determines whether the carrier has banned the requested application from downloading (globally or targeted based upon other criteria such as subscriber identity, device type, etc.). In step 1601, the routine requests and obtains, a list of applications that the carrier has declined to allow to be downloaded. This list may be retrieved locally and updated on a periodic basis, for example using the MAS Query Engine 650 in Figure 6. In step 1602, the routine searches the retrieved list for the requested application to determine if the application is banned. This provides a quick and robust way to prohibit the downloading of applications that, for example, contain or are suspected of containing malicious code. This method provides a centrally based approach (as compared to a distributed approach where each device obtains a "virus checker" and a malicious application datafile) to stop the spread of malicious applications. In step 1603, the routine determines if the request is for a banned application and, if so, proceeds to step 1605, else proceeds to step 1604. In step 1604, the request is logged, and the routine returns with successful status. In step 1605, the failed request is logged, a notification is sent to the subscriber, and the routine returns a failed status.

Figure 17 is an example flow diagram of processing performed by a Verify Subscriber routine of a Provisioning Manager. (*See* step 1503 of Figure 15.) In summary, the Verify Subscriber routine compares subscriber profiles to content categories and service plan definitions, as stored and implemented by the Administrator component in profiles (*e.g.*, Administrator 509 in Figure 5), and determines whether the subscriber is authorized to download the requested application. Specifically, in step 1701, the routine determines from which carrier the request message was received. In step 1702, the routine identifies the subscriber who sent the request. This may be accomplished, for example, by examining the request message for routing information. In step 1703, the routine establishes a connection with the determined carrier if subscriberprofile information is stored on the carrier, and in step 1704, retrieves the identified subscriber's profile from the carrier. One skilled in the art will appreciate that the subscriber's profile may also be stored locally on and retrieved from the MAS using, for example, the Profile Reader 652 component in Figure 6 to access a local data repository 511 in Figure 5. In step 1705, the routine examines the request to determine which application has been requested. In step 1706, the routine determines if the subscriber's profile authorizes downloading of the requested application. This determination can be accomplished, for example, by examining the service plan of the subscriber group to which the subscriber belongs to determine whether the application belongs to a content category associated with that service plan. In addition, the routine may check for the presence of a matching banned application in the subscriber profile, and if a match is found, subsequently

reject the request. In step 1707, if it is determined that the request is authorized, then the routine proceeds to step 1708, else it proceeds to step 1709. In step 1708, the request is logged and the routine returns with a successful status. In step 1709, the failed request is logged, a notification sent to the subscriber, and the routine returns with a failure status.

Figure 18 is an example flow diagram of processing performed by a Verify Device routine of a Provisioning Manager. (*See* step 1504 of Figure 15.) In summary, the Verify Device routine compares the device profile associated with the subscriber's device to the application profile for the requested application and verifies that the resources required by the application are available according to the device profile. In step 1801, the routine identifies the type of subscriber device from which the request was received. One skilled in the art will recognize that this information is determined in protocol negotiations and typically may be extracted from routing information stored in the request message. In step 1802, the routine determines the capabilities of the subscriber device by accessing a previously stored device profile that is associated with the identified device. In one embodiment, the device profile is retrieved using the Profile Reader 652 in Figure 6. If a device profile is not found for the identified device, the event is logged and the system administrator is notified accordingly. (*In one embodiment*, carriers are made aware of the particular kind of device used by each subscriber when the subscriber registers with the carrier to obtain a phone number; carriers should preferably ensure that all registered device types are supported with a device profile.) The device profile contains information relevant to the capabilities of the subscriber device such as memory capacity, processor type, processing speed, maximum size of a downloadable application, etc. In step 1803, the routine determines the requirements for the requested application by retrieving and examining the application profile that corresponds to the requested application, as previously created by the Administrator component. The application profile contains the requirements for executing the application including, for example, the amount of memory required, API calls made, and minimal processor speed. The requirements may also be specified in the application profile according to types of subscriber devices that are supported. In step 1804, the capabilities of the device are compared to the requirements of the requested application by comparing the device and application profiles. In step 1805, the routine determines if the device is capable of running the requested application and, if so, proceeds to step 1806, else it proceeds to step 1807. In step 1806, the request is logged and the routine returns with a successful status. In step 1807, the failed request is logged, a notification is sent to the subscriber, and the routine returns with a failure status.

Figure 19 is an example flow diagram of processing performed by a Perform Open Provisioning routine of a Provisioning Manager. (*See* step 1407 of Figure 14.) In steps 1901 and 1902, the Provisioning Manager needs to determine whether there is a provisioned application already available or cached and, if so, continues in step 1903, else continues in step 1904. This scenario could occur, for example, if the application, even though it is from an untrusted or unknown source, has been requested and provisioned before. In step 1903, the provisioned application is returned. Alternatively, if no preprovisioned application is found, then in step 1904, the routine retrieves the application using the designated URL provided in the request message. This application may have been processed before by the MAS, and thus may have a corresponding application profile already. Thus, in step 1905, the routine determines whether a corresponding application profile exists and, if so continues in step 1907, else creates a new application profile in step 1906, and then continues in step 1907. In step 1907, the routine performs device verification by comparing the application profile (retrieved or created) to a device profile that corresponds to the device type of the subscriber's request. In step 1908, the routine invokes the provisioning interface of the Deployment Manager to provision an untrusted application, and in step 1909 returns the provisioned application.

Figure 20 is an example flow diagram of processing performed by a Perform Application Discovery routine of a Provisioning Manager. (*See* step 1409 of Figure 14.) There are two basic types of application discovery: a search for applications that match a criteria specified by the subscriber and a system provided application list based upon stored subscriber preferences. Specifically, in step 2001, the routine determines whether the user has designated search criteria and, if so, continues in step 2002, else continues in the2004. In step 2002, the routine searches the application store (a data repository of applications) and queries for applications (content) that matches the designated criteria. Example criteria include category, price, gender, age, etc. In step 2003, the list is initially set to these query results, and the routine continues in step 2007. In step 2004, the routine determines whether there is a Personal Access List (a "PAL") available and, if so, continues in step 2005 to set the

list initially to the determined PAL, else continues in step 2006 to set the list initially to a default value. In step 2007, the MAS adds a set of defined applications to the initial list, known as the startdeck. The startdeck essentially allows the MAS to reserve slots in the application discover sessions, for example, for higher revenue advertisers. In step 2008, the routine invokes the Verify Subscriber routine, as discussed with respect to Figure 17, for each application initially on the list. Any applications not passing any one of the filtration steps 2008-2009 will be filtered from the list before the next step in the process. In step 2009, the routine invokes the Verify Device, as discussed with respect to Figure 18, for each application initially on the list. In step 2010, the routines generates an XML for internal standardized format and in step 2011, transforms the contents of this list to an appropriate language that corresponds to the subscriber device.

Figure 21 is an example flow diagram of processing performed by a Deployment Manager of a Mobile Application System to provide provisioned applications in response to requests by subscribers and system administrators. (*See*, for example, Deployment Manager 506 in Figure 5.) System administrators may request that popular applications for popular devices be pre-provisioned (statically provisioned) and cached for the purpose of minimizing the time required to respond to a subscriber's request. Alternatively, all applications may be dynamically provisioned, and optionally cached. In step 2101, the Deployment Manager is initialized. In step 2102, the Deployment Manager evaluates a request to determine the identity of the requested application. In step 2103, the Deployment Manager invokes the Procure Provisioned Application routine to control the retrieval of the content and to cause provisioning to occur, as described further with reference to Figure 22. In step 2104, the Deployment Manager determines whether the request is made by a system administrator to initiate storing of the provisioned application and, if so, proceeds to step 2105, else proceeds to step 2106. In step 2105, the Deployment Manager stores the provisioned application in the cache, the carrier's application store, or a remote application host's server, depending on the policy of the system administrator, and ends processing. In step 2106, the Deployment Manager sends the provisioned application to the Provisioning Manager, and then ends processing.

Figure 22 is an example flow diagram of processing performed by a Procure Provisioned Application routine of a Deployment Manager. (*See* step 2105 in Figure 21.) In summary, the Deployment Manager retrieves the application code and inspects, optimizes, and instruments it according to current policies implemented in the MAS. In step 2201, the routine consults some type of index to determine whether a pre-provisioned version of the application exists in a location known to the MAS. The manner in which this information is stored is related to how the cache and/or data repositories are implemented. Well-known techniques for implementing a cache using a local fast data store and index may be used. Applications may be pre-provisioned and stored when it is expected that large numbers of requests will be made for an application that would entail the same provisioning requirements. This may occur, for example, when large numbers of users who have the same kind of subscriber device request the same application. In such cases, the application may be provisioned and stored in the cache (and retrieved when requests are made by users having subscriber devices for which the application was provisioned) or stored in other MAS data repositories. In step 2202, if a pre-provisioned version of the application exists, then the routine proceeds to step 2203, else it proceeds to step 2207. In step 2203, the location of the pre-provisioned application is determined. In step 2204, the routine determines if the pre-provisioned application is stored locally and, if so, proceeds to step 2205, else it proceeds to step 2206. In step 2205, the routine fetches the application locally (typically from the carrier's application store, which may be located in the MAS or on the carrier's premises), and returns. In step 2206, the routine fetches the application from a remote application host (*e.g.*, a third-party server) and returns. If, on the other hand, in step 2202, the routine determines that a pre-provisioned version of the requested application does not exist, then in step 2207 the routine determines the location of the unprocessed, un-provisioned application. In step 2208, the routine determines if the application code is stored locally and, if so, proceeds to step 2209, else proceeds to step 2210. In step 2209, the routine fetches the application code from the carrier's application store or other local storage. In step 2210, the routine fetches the application code from a remote application host. In step 2211, the routine provisions the fetched application, as described further with reference to Figure 23, and returns.

Figure 23 is an example flow diagram of processing performed by a Provision Application routine of a Deployment Manager. In step 2301, the Provision Application routine inspects the application as described further with reference to Figure 24. In step

2302, the routine optimizes the application as described further with reference to Figure 25. In step 2303, the routine installs instrumentation in the application as described further with reference to Figure 26. In step 2304, the routine packages the application in a format suitable for delivery as described further with reference to Figure 23, and returns.

Figure 24 is an example flow diagram of processing performed by an Inspect Application routine of a Deployment Manager. (*See*, for example, step 2301 in Figure 23.) In step 2401, the routine deconstructs/decodes the structure of the application code if required to identify APIs, including packages, classes, methods, and fields, or other structures as appropriate. When the applications are coded in Java, then inspection can be performed against the binary program, with no need to insert source code level checks in the application itself to generated debugging/logging information. A set of inspection steps is described as examples in steps 2401-2405; however, one skilled in the art will recognize that other inspection steps, in addition to or instead of the ones described herein may be applied as appropriate. In step 2402, the routine retrieves any application filters that are relevant for the potential targets under examination (the requested application, the requesting subscriber, the content provider of the application, and global filters). In one embodiment, these filters are stored in a MAS data repository, however they could be stored in any known location. In step 2403, the routine inspects the retrieved application for malicious and banned code by comparing the deconstructed code with stored indications of banned data structures and API as described by the retrieved application filters. In step 2404, the routine determines the number, type, and frequency of API calls present in the code and checks whether they meet the system administrator requirements, which may be stored in the application filters. In step 2405, the routine performs a flow analysis of the deconstructed application and determines whether the number of threads activated are within the requirements of the system administrators. This flow analysis can be accomplished using techniques such as creating a directional graph of the code and applying well-known graph analysis algorithms. One skilled in the art will recognize that other checks may also be performed on the retrieved application. In step 2406, the routine determines if the retrieved application has passed inspection and, if so, returns a success status, otherwise the routine flags the failed condition and returns a failure status.

Figure 25 is an example flow diagram of processing performed by an Optimize Application routine of a Deployment Manager. (*See*, for example, step 2302 in Figure 23.) One skilled in the art will recognize that any well-known code optimization techniques can be incorporated in this routine, and what is shown is an example. In step 2501, the routine shortens variable names contained in the retrieved application for the purpose of shortening the file size of the requested application. In step 2502, the routine maps the executable paths of the retrieved application. In step 2503, the routine removes any unused code for the purpose of shortening the file length, and continues with similar optimization steps. When finished optimizing, the routine returns.

Figure 26 is an example flow diagram of processing performed by an Install Instrumentation routine of a Deployment Manager. (*See*, for example, step 2303 in Figure 23.) In step 2601, the routine retrieves the identified subscriber's profile from typically a local data repository using, for example, the Profile Reader 652 in Figure 6. In step 2602, the routine determines the carrier's policy for the identified subscriber when using the requested application. For example, certain subscribers may be allowed to use the application on a subscription basis or even a trial basis, but others may not be allowed. As discussed above with reference to Figure 7, the instrumentation implements certain policies. For example, it can provide a code wrapper that allows the provisioned code to be executed a limited number of times or during a given period in time. In step 2603, the routine installs the instrumentation in the requested application according to the determined carrier's policy, after which the routine returns. In an example embodiment, the Install Instrumentation routine uses byte code instrumentation techniques to insert new code or to modify existing code within the application at the binary level. The code to be instrumented may be provided directly by the Install Instrumentation routine, or may be retrievable from other data storage, for example data storage associated with different carrier policies.

Figure 27 is an example flow diagram of processing performed by a Package Application routine of a Deployment Manager. (*See*, for example, step 2304 in Figure 23.) In step 2701, the routine accesses the retrieved subscriber device profile to determine compatible file formats for the identified subscriber device. In step 2702, the routine determines whether the subscriber device is capable of reading compressed files and, if so, proceeds to step 2703, else proceeds to step 2704. In step 2703, the routine compresses the

provisioned application for the purpose of minimizing transmission time and the number of bytes transmitted. In step 2704, the routine packages the application using a determined file format by encapsulating the provisioned application with information sufficient to enable the Handset Administration Console (*See*, for example, the Handset Administration Console of Figure 2) executing on a wireless device to extract the application. As described earlier, one format preferred by many Java-enabled wireless devices is compressed JAR files. In some cases, however, the application needs to be distributed to the device in smaller packets, which are reassembled on the wireless device for installation. The Billing Manager, discussed below with reference to Figure 28, also relies on the encapsulating information for billing and routing purposes. After the application has been packaged, the routine returns.

Figure 28 is an example flow diagram of processing performed by a Billing Manager of a Mobile Application System. (*See*, for example, Billing Manager 507 in Figure 5.) In step 2801, the Billing Manager is initialized. In step 2802, the Billing Manager determines if it is time to generate a billing report and, if so, proceeds to step 2803, else proceeds to step 2804. In an alternative embodiment, the Billing Manager may respond to an administrative query, for example, from the Administrator component, to generate a billing report. In step 2803, the Billing Manager generates a billing report based on parameters set by a system administrator. In step 2804, the Billing Manager determines whether there is a request to log provisioning information (for billing purposes) and, if so, proceeds to step 2805, else it returns. In step 2805, the Billing Manager logs parameters of the request that relate to billing (*e.g.*, the identity of the user making the request, the category of the request, the size of the download required, etc.) and the status of system variables (*e.g.*, the date, time of day, etc.) to be used for future billing. For example, the length of the application, the time at which the application was requested, the time required to process the application, and the number of applications may be used in generating a billing report. In addition, if prepaid billing is supported, then the Billing Manager may generate an accounting request to the carrier to properly decrement the subscriber's prepaid billing account. After the billing report has been generated and the appropriate parameters logged, the Billing Manager returns.

D:\NrPortbl\iManage\ANGIEL\261942_1.DOC

APPENDIX E

## METHOD AND SYSTEM FOR PACKET LEVEL BILLING IN WIRELESS APPLICATION ENVIRONMENTS

### TECHNICAL FIELD

The present invention relates to a method and system for wireless application environments and, in particular, to methods and systems for billing and logging information at the packet level in wireless applications.

### DETAILED DESCRIPTION OF THE INVENTION

Embodiments of the present invention provide computer- and network-based methods and systems for packet level billing and other types of statistics gathering and logging at the packet level in a wireless application environment (a "packet level billing system"). Several example embodiments are described in detail in Appendix A, which is herein incorporated by reference in its entirety. Although the methods and systems are described for use in generating and logging billing information so that airtime is not the only statistic used to determine billing for wireless applications, one skilled in the art will recognize that any of these mechanisms can be used for logging other types of statistical information, such as other time and usage information. For use with other statistics, instead of providing a server-side implementation that implements the policies of a billing server, any type of data collection policy may be instead implemented by the specifications, API, and / or other code described in Appendix A-E.

The packet level billing system may be implemented in an intrusive or non-intrusive manner, thus allowing a varied degree of flexibility in both what part of the code is changed to include the generation of packet level billing and in what code is aware that the packet level billing generation code is being invoked. Also, several of the methods can collect and generate the billing (or any other type of logging information) transparently to the network application developer. The methods and systems include a "Writing to Specification" solution, a Library solution, a Device Software (driver) Modification solution, and an Instrumentation solution.

From the foregoing it will be appreciated that, although specific embodiments of the invention have been described herein for purposes of illustration, various modifications may be made without deviating from the spirit and scope of the invention. For example, one skilled in the art will recognize that the methods and systems discussed herein are applicable to the gathering and generating of any type of application data. One skilled in the art will also recognize that the methods and systems discussed herein are applicable to differing protocols, communication media (optical, wireless, cable, and the like) and client devices (such as wireless handsets, electronic organizers, portable email machines, game machines, pagers, navigation devices such as GPS receivers, and the like).

L:\320037 - 4thpass\403P1\403P1.doc

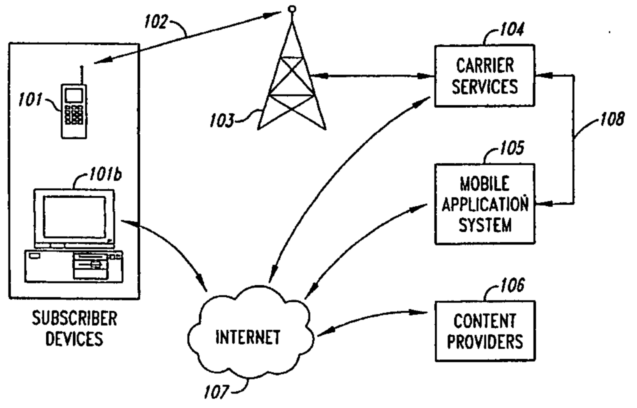

102
101
103
104
CARRIER SERVICES
108
105
MOBILE APPLICATION SYSTEM
101b
106
CONTENT PROVIDERS
SUBSCRIBER DEVICES
INTERNET
107

Fig. 1

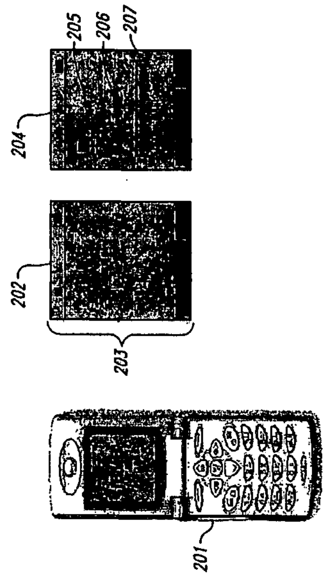

205
206
207
204
202
203
201

Fig. 2

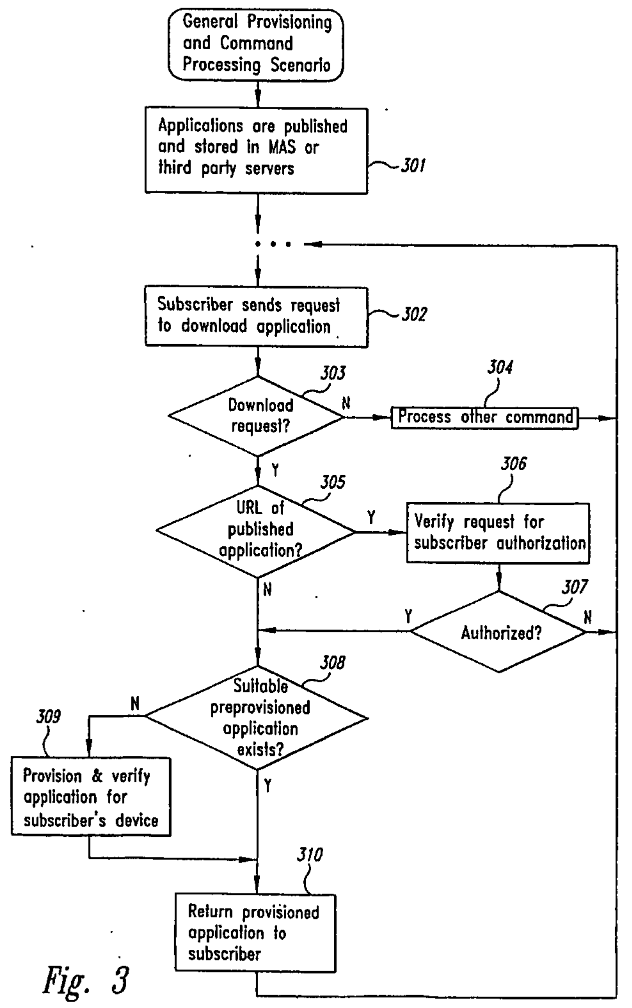

General Provisioning and Command Processing Scenario
Applications are published and stored in MAS or third party servers
301
Subscriber sends request to download application
302
Download request?
303
N
Y
Process other command
304
URL of published application?
305
Y
N
Verify request for subscriber authorization
306
Authorized?
307
Y
N
Suitable preprovisioned application exists?
308
N
Y
Provision & verify application for subscriber's device
309
Return provisioned application to subscriber
310

Fig. 3

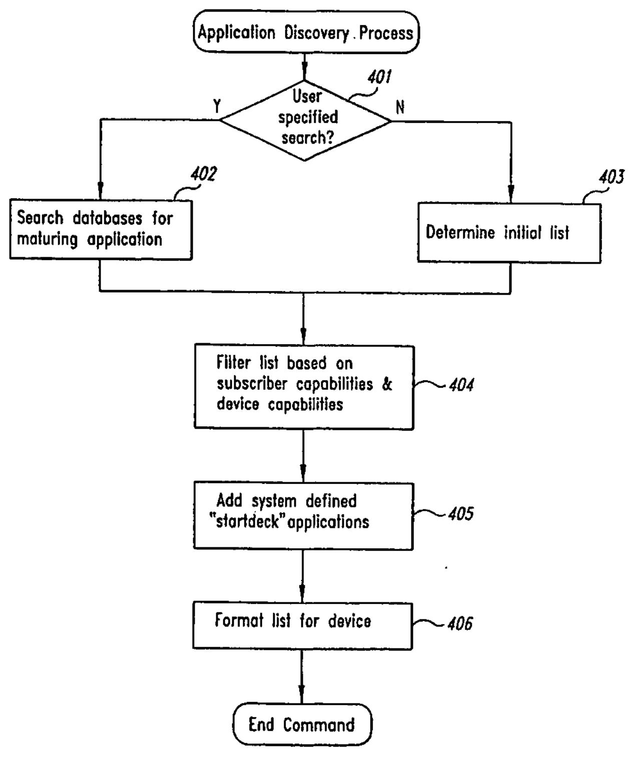

Application Discovery Process
User specified search?
401
Y
N
Search databases for maturing application
402
Determine initial list
403
Filter list based on subscriber capabilities & device capabilities
404
Add system defined "startdeck" applications
405
Format list for device
406
End Command

Fig. 4

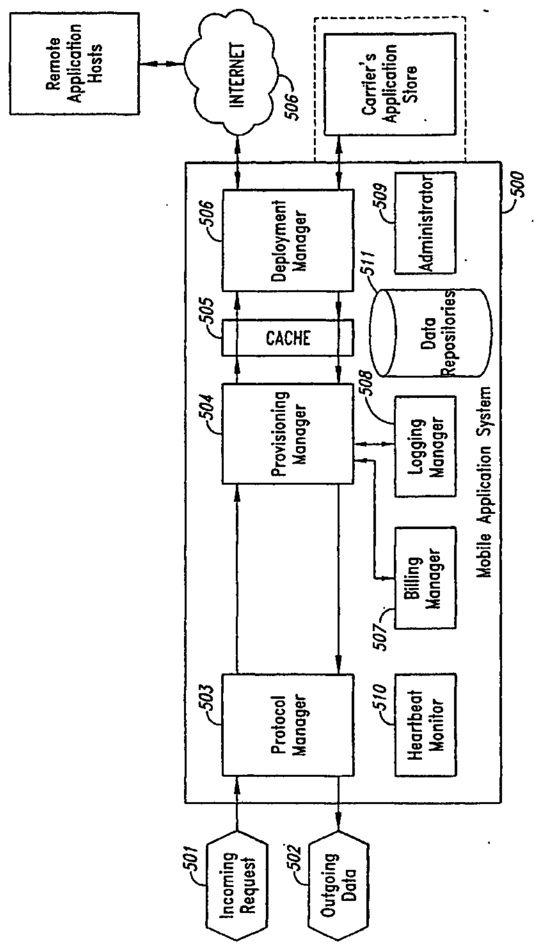

Remote Application Hosts
INTERNET
506
Carrier's Application Store
Mobile Application System
500
506
Deployment Manager
509
Administrator
505
CACHE
511
Data Repositories
504
Provisioning Manager
508
Logging Manager
507
Billing Manager
503
Protocol Manager
510
Heartbeat Monitor
501
Incoming Request
502
Outgoing Data

*Fig. 5*

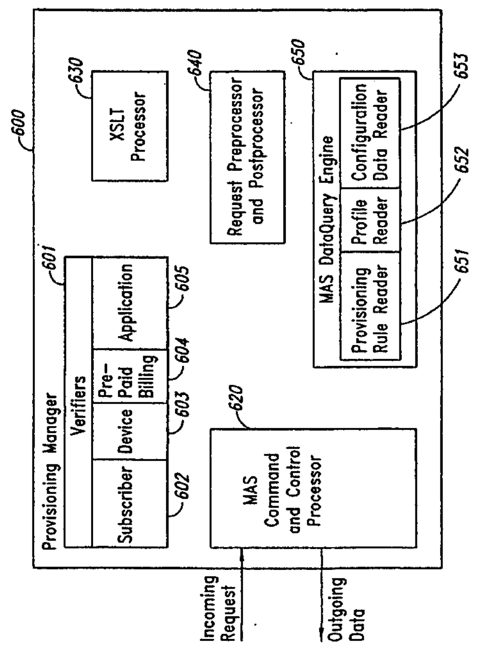

600
630
XSLT Processor
640
Request Preprocessor and Postprocessor
650
MAS DataQuery Engine
Configuration Data Reader
653
Profile Reader
652
Provisioning Rule Reader
651
Provisioning Manager
601
Verifiers
Application
605
Pre-Paid Billing
604
Device
603
Subscriber
602
620
MAS Command and Control Processor
Incoming Request
Outgoing Data

*Fig. 6*

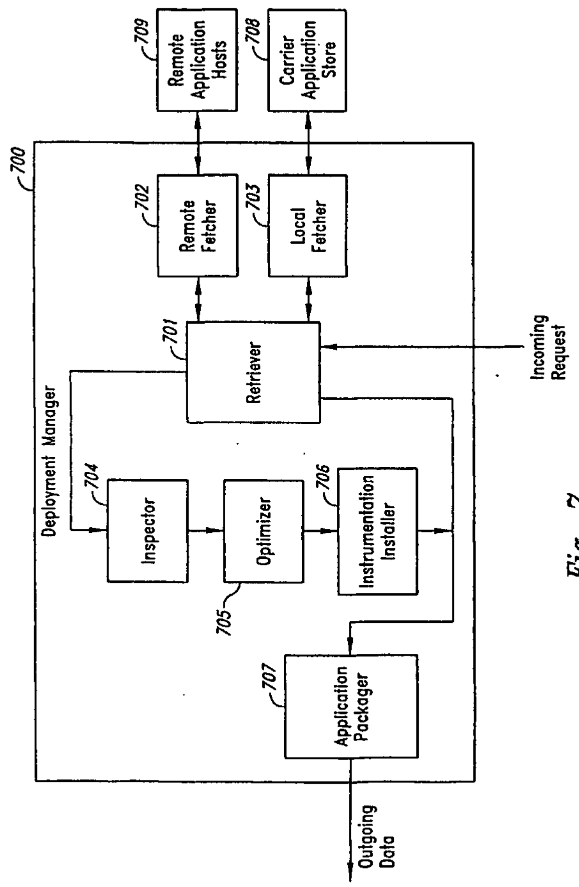

700
Deployment Manager
701
Retriever
702
Remote Fetcher
703
Local Fetcher
704
Inspector
705
Optimizer
706
Instrumentation Installer
707
Application Packager
708
Carrier Application Store
709
Remote Application Hosts
Incoming Request
Outgoing Data

Fig. 7

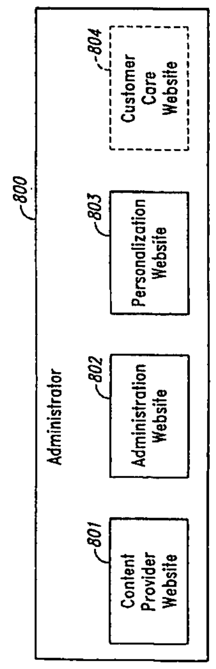
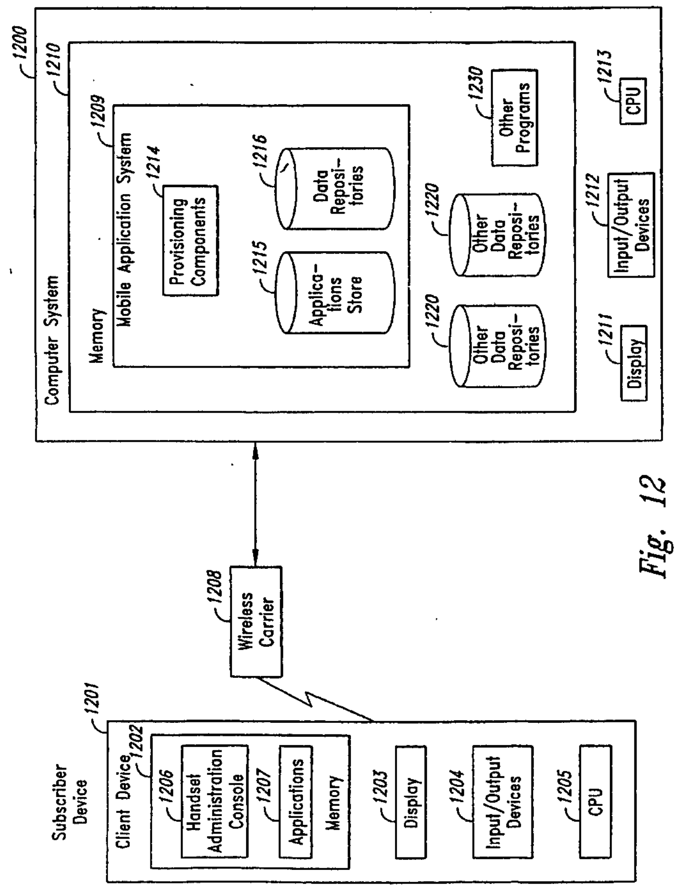
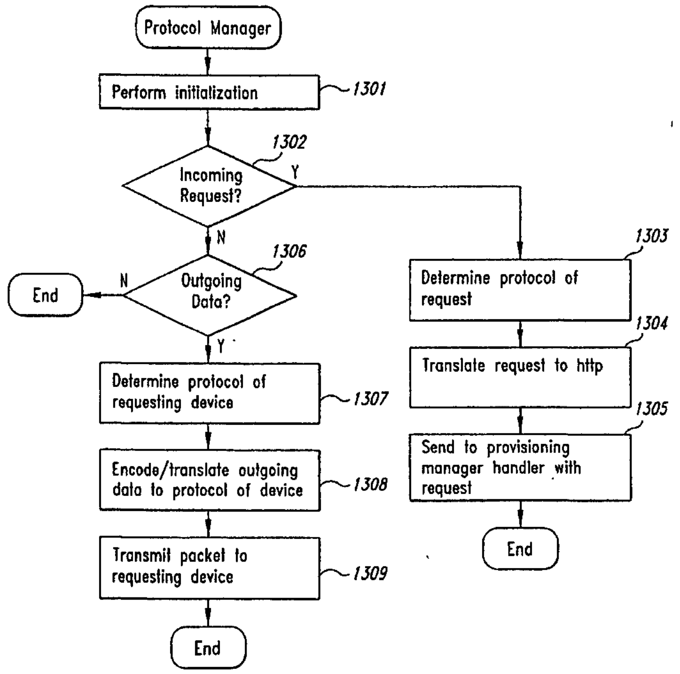

800
Administrator
801
Content Provider Website
802
Administration Website
803
Personalization Website
804
Customer Care Website

Fig. 8



1200
Computer System
1210
Memory
1209
Mobile Application System
1214
Provisioning Components
1215
Applica-tions Store
1216
Data Reposi-tories
1220
Other Data Reposi-tories
1220
Other Data Reposi-tories
1230
Other Programs
1211
Display
1212
Input/Output Devices
1213
CPU
1208
Wireless Carrier
1201
Subscriber Device
Client Device
1202
Memory
1206
Handset Administration Console
1207
Applications
1203
Display
1204
Input/Output Devices
1205
CPU

Fig. 12



Protocol Manager
Perform initialization
1301
Incoming Request?
1302
Y
N
Outgoing Data?
1306
N
Y
End
Determine protocol of requesting device
1307
Encode/translate outgoing data to protocol of device
1308
Transmit packet to requesting device
1309
End
Determine protocol of request
1303
Translate request to http
1304
Send to provisioning manager handler with request
1305
End

Fig. 13

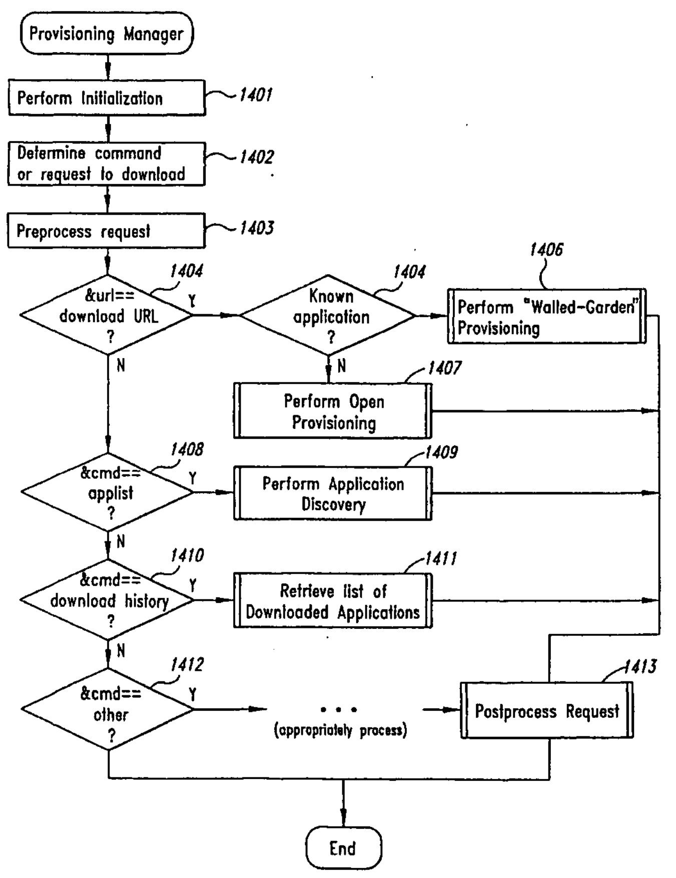

Provisioning Manager
Perform Initialization
1401
Determine command or request to download
1402
Preprocess request
1403
&url== download URL ?
1404
Y
N
Known application ?
1404
N
Perform "Walled-Garden" Provisioning
1406
Perform Open Provisioning
1407
&cmd== applist ?
1408
Y
N
Perform Application Discovery
1409
&cmd== download history ?
1410
Y
N
Retrieve list of Downloaded Applications
1411
&cmd== other ?
1412
Y
• • • (appropriately process)
Postprocess Request
1413
End

Fig. 14

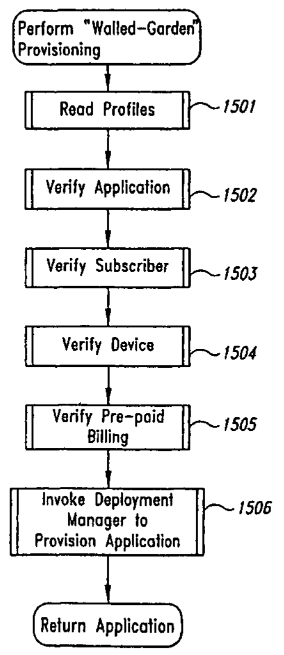

Perform "Walled-Garden" Provisioning
Read Profiles
1501
Verify Application
1502
Verify Subscriber
1503
Verify Device
1504
Verify Pre-paid Billing
1505
Invoke Deployment Manager to Provision Application
1506
Return Application

Fig. 15

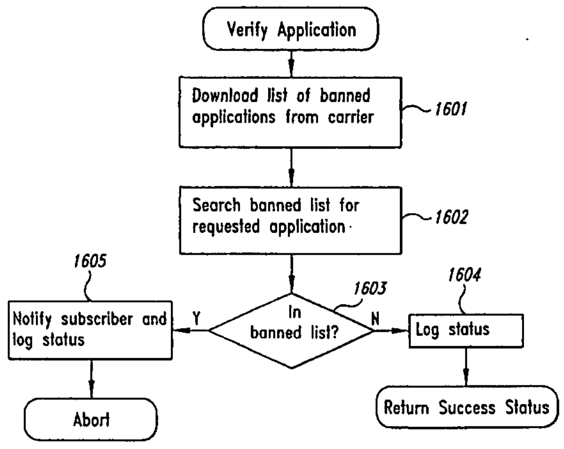

Verify Application
Download list of banned applications from carrier
1601
Search banned list for requested application
1602
In banned list?
1603
Y
N
Notify subscriber and log status
1605
Abort
Log status
1604
Return Success Status

Fig. 16

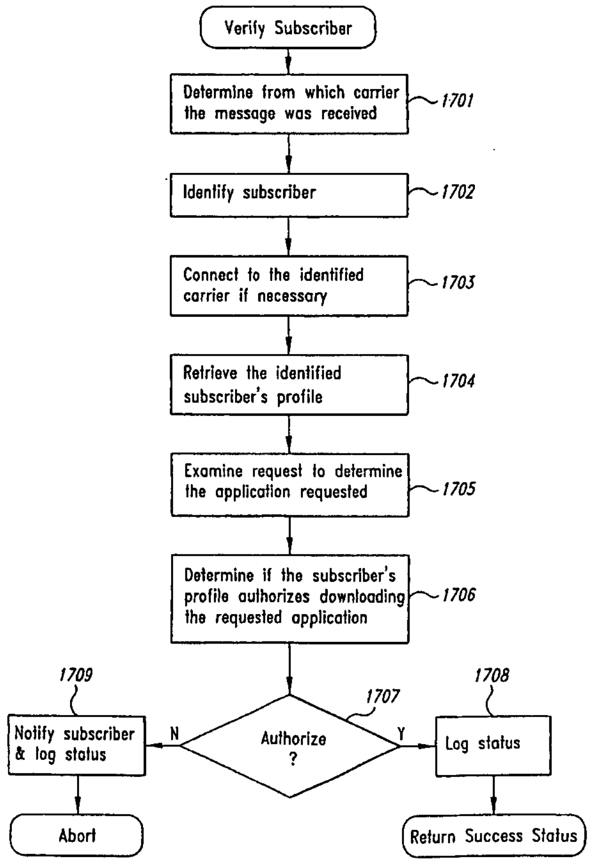

Verify Subscriber
Determine from which carrier the message was received
1701
Identify subscriber
1702
Connect to the identified carrier if necessary
1703
Retrieve the identified subscriber's profile
1704
Examine request to determine the application requested
1705
Determine if the subscriber's profile authorizes downloading the requested application
1706
Authorize ?
1707
N
Y
Notify subscriber & log status
1709
Abort
Log status
1708
Return Success Status

Fig. 17

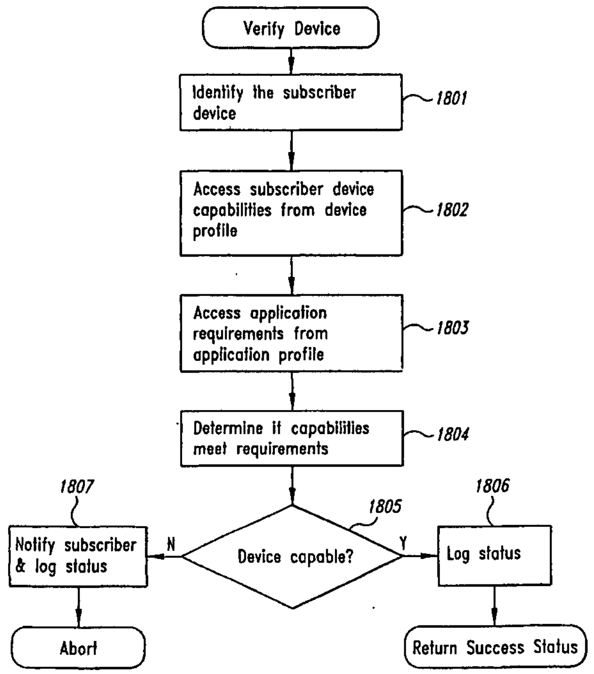

Verify Device
Identify the subscriber device
1801
Access subscriber device capabilities from device profile
1802
Access application requirements from application profile
1803
Determine if capabilities meet requirements
1804
Device capable?
1805
Y
N
Log status
1806
Return Success Status
Notify subscriber & log status
1807
Abort

*Fig. 18*

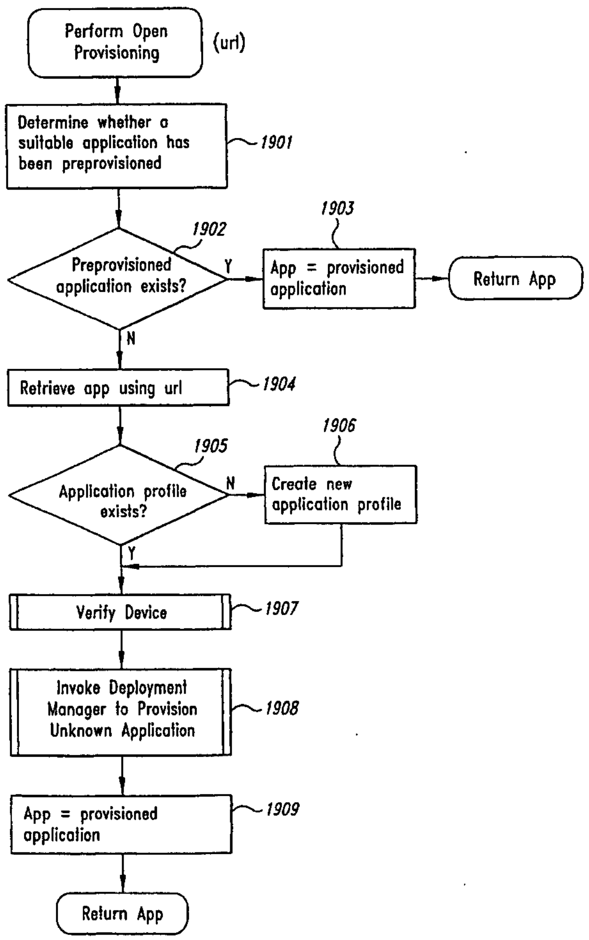
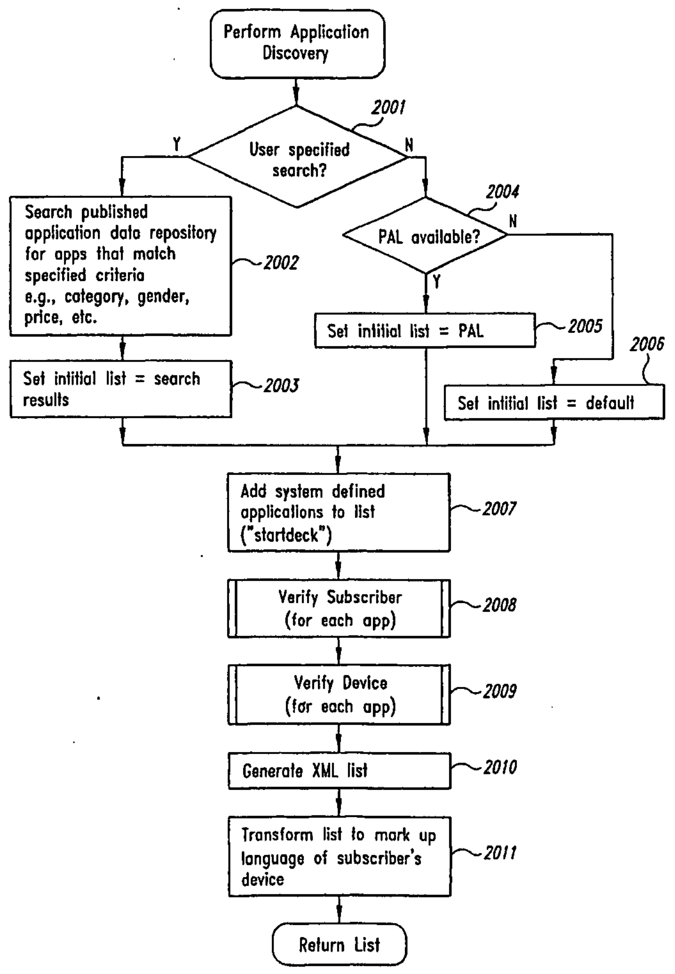
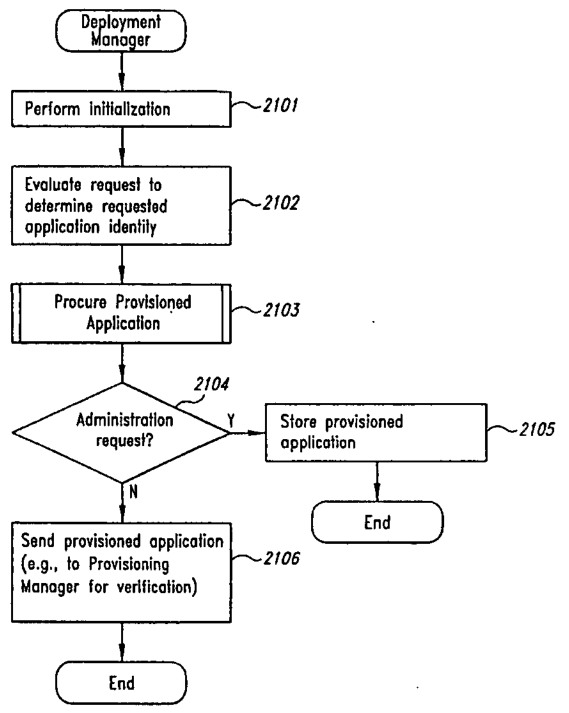

Perform Open Provisioning
(url)
Determine whether a suitable application has been preprovisioned
1901
Preprovisioned application exists?
1902
Y
N
App = provisioned application
1903
Return App
Retrieve app using url
1904
Application profile exists?
1905
N
Y
Create new application profile
1906
Verify Device
1907
Invoke Deployment Manager to Provision Unknown Application
1908
App = provisioned application
1909
Return App

*Fig. 19*



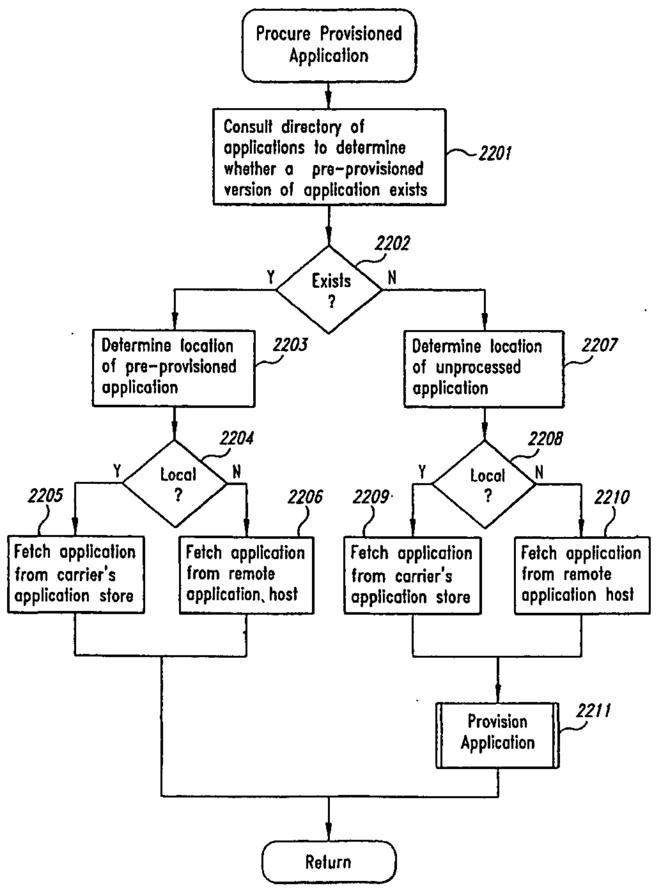
Perform Application Discovery
User specified search?
2001
Y
N
Search published application data repository for apps that match specified criteria e.g., category, gender, price, etc.
2002
PAL available?
2004
Y
N
Set initial list = PAL
2005
2006
Set initial list = search results
2003
Set initial list = default
Add system defined applications to list ("startdeck")
2007
Verify Subscriber (for each app)
2008
Verify Device (for each app)
2009
Generate XML list
2010
Transform list to mark up language of subscriber's device
2011
Return List

Fig. 20



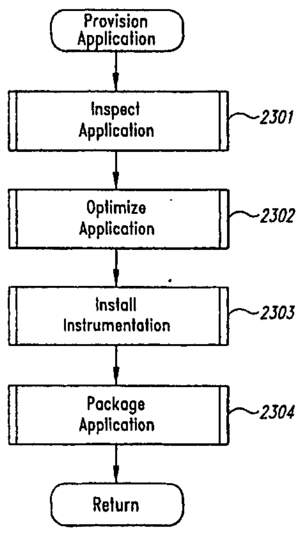
Deployment Manager
Perform initialization
2101
Evaluate request to determine requested application identity
2102
Procure Provisioned Application
2103
2104
Administration request?
Y
N
Store provisioned application
2105
End
Send provisioned application (e.g., to Provisioning Manager for verification)
2106
End

Fig. 21



Procure Provisioned Application
Consult directory of applications to determine whether a pre-provisioned version of application exists
2201
Exists ?
2202
Y
N
Determine location of pre-provisioned application
2203
Determine location of unprocessed application
2207
Local ?
2204
Y
N
Local ?
2208
Y
N
2205
Fetch application from carrier's application store
2206
Fetch application from remote application, host
2209
Fetch application from carrier's application store
2210
Fetch application from remote application host
Provision Application
2211
Return

Fig. 22



Provision Application
Inspect Application
2301
Optimize Application
2302
Install Instrumentation
2303
Package Application
2304
Return

Fig. 23

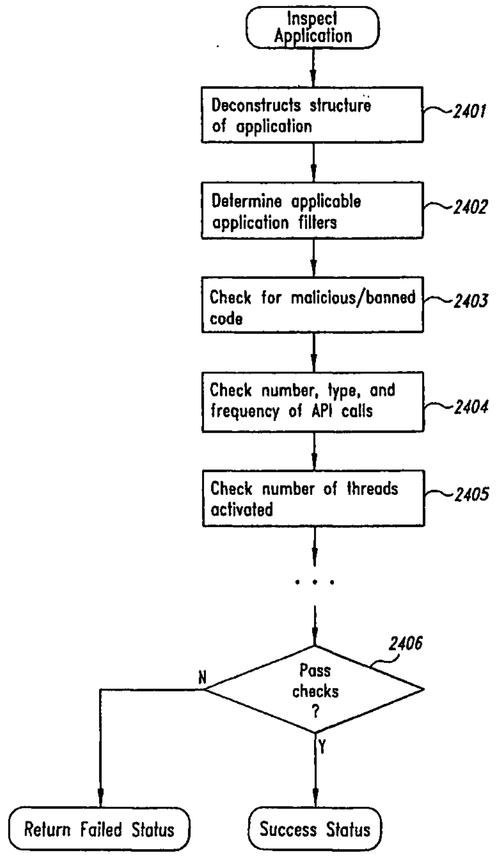

Inspect Application
Deconstructs structure of application
2401
Determine applicable application filters
2402
Check for malicious/banned code
2403
Check number, type, and frequency of API calls
2404
Check number of threads activated
2405
2406
Pass checks ?
N
Y
Return Failed Status
Success Status

Fig. 24

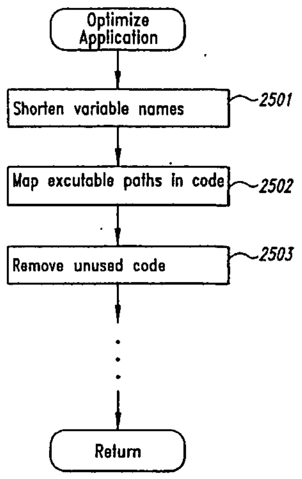


Fig. 25

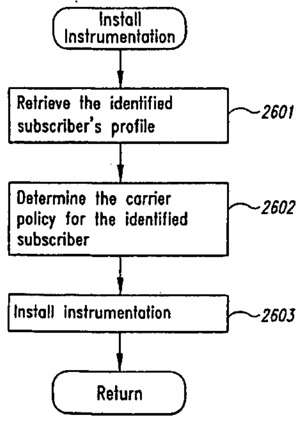

Install Instrumentation
Retrieve the identified subscriber's profile
2601
Determine the carrier policy for the identified subscriber
2602
Install instrumentation
2603
Return

Fig. 26

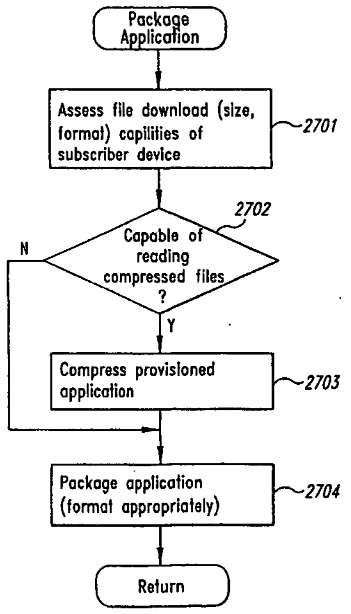

Package Application
Assess file download (size, format) capilities of subscriber device
2701
Capable of reading compressed files ?
2702
N
Y
Compress provisioned application
2703
Package application (format appropriately)
2704
Return

Fig. 27

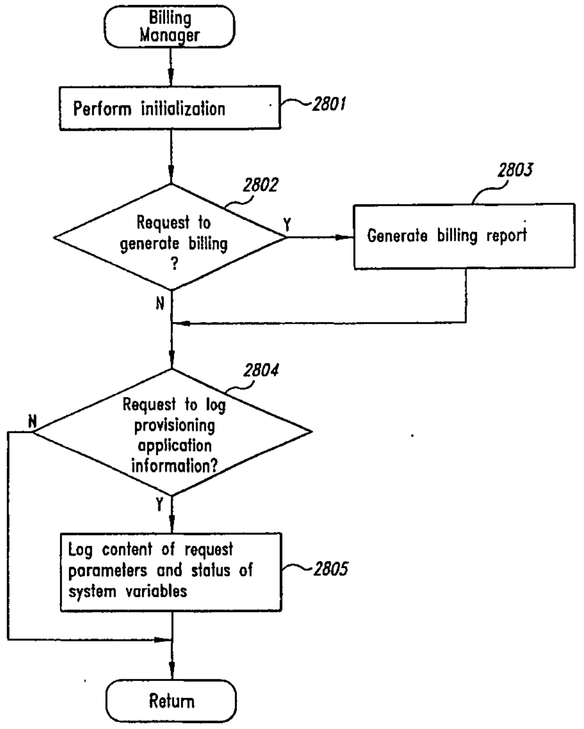

Billing Manager
Perform initialization
2801
Request to generate billing ?
2802
Y
N
Generate billing report
2803
Request to log provisioning application information?
2804
N
Y
Log content of request parameters and status of system variables
2805
Return

Fig. 28

## CLAIMS

1. A method in a computer-based environment for providing transmission-based billing of content that transmits data over a network, comprising:

determining billing tracking code; and

instrumenting the determined billing tracking code into the content thereby modifying the content, such that, when the modified content is executed on a target device, the billing tracking code automatically communicates billing data based upon an amount of data transmitted between the modified content and the network.

2. The method of claim 1 wherein the billing tracking code tracks the amount of data sent from the instrumented content over the network.

3. The method of claim 2 wherein the network is the Internet.

4. The method of claim 2 wherein the amount of data is tracked at a packet level that is logically defined.

5. The method of claim 1 wherein the billing tracking code tracks the amount of data received by the instrumented content from the network.

6. The method of claim 5 wherein the network is the Internet.

7. The method of claim 5 wherein the amount of data is tracked at a packet level that is logically defined.

8. The method of claim 1 wherein the content is Java-based.

9. The method of claim 1 wherein the content contains byte-code instructions.

10. The method of claim 1 wherein the instrumenting is accomplished at a byte-code level of content examination.

11. The method of claim 1 wherein the instrumented content includes a security key and wherein the security key is transmitted with the automatically communicated billing data so that the integrity of the source of the billing data can be verified.

12. The method of claim 11 wherein the security key is based upon a random number.

13. The method of claim 11 wherein the security key is application and subscriber specific.

14. The method of claim 11 wherein the security key in instrumented into the content upon receiving a request to download the content.

15. The method of claim 1 wherein the environment is integrated with a wireless carrier infrastructure.

16. The method of claim 1, further comprising causing the instrumented content to be downloaded to a target device over a wireless transmission medium.

17. The method of claim 16 wherein the content is requested by a subscriber of a carrier from the computer-based environment over a wireless transmission medium.

18. The method of claim 1, further comprising causing the instrumented content to be downloaded to a target device over a wired transmission medium.

19. The method of claim 18 wherein the wired transmission medium is the Internet.

20. The method of claim 1 wherein the billing data comprises at least one of amount of data sent, amount of data received, a time stamp, an application identifier, a security key, a transaction identifier, and a retry expiration indicator.

21. The method of claim 1 wherein the billing data is automatically communicated on the transmission basis to a billing server system.

22. The method of claim 21 further comprising transmitting data that is not billing data directly between the subscriber device and a server system that is not the billing server system.

23. The method of claim 1, further comprising integrating the billing data with customer-based billing information to generate a customer data record.

24. The method of claim 1 wherein the billing data is used to support a plurality of billing policies.

25. The method of claim 24 wherein the billing policies include a promotional offer that provides reduced charges for a designated application.

26. The method of claim 24 wherein the billing policies are provided by a content provider.

27. The method of claim 1 wherein the billing data are used to provide royalty payments to providers of the content.

28. The method of claim 1, further comprising causing data transmitted between the instrumented content and the network to be routed in accordance with the automatically communicated transmission based billing data.

29. The method of claim 28 wherein the routing enables efficient use of resources on the network.

30. The method of claim 28 wherein a priority is assigned to the content based upon transmission usage.

31. The method of claim 1 wherein the billing tracking code utilizes a proxy store and forward technique to transmit billing data and data transmission packets between the instrumented content and a plurality of server systems.

32. A network-based transmission medium containing content that has been instrumented with billing tracking code, whereby the billing tracking code automatically generating billing data on a transmission basis when the content is executed on a target device.

33. The transmission medium of claim 32 wherein the billing tracking code tracks the amount of data sent from the instrumented content over the network.

34. The transmission medium of claim 33 wherein the network is the Internet.

35. The transmission medium of claim 33 wherein the amount of data is tracked at a packet level that is logically defined.

36. The transmission medium of claim 32 wherein the billing tracking code tracks the amount of data received by the instrumented content from the network.

37. The transmission medium of claim 36 wherein the network is the Internet.

38. The transmission medium of claim 36 wherein the amount of data is tracked at a packet level that is logically defined.

39. The transmission medium of claim 32 wherein the content is Java-based.

40. The transmission medium of claim 32 wherein the instrumenting is accomplished at a byte-code level of content examination.

41. The transmission medium of claim 32 wherein the content contains byte-code instructions.

42. The transmission medium of claim 32 wherein the instrumented content includes a security key.

43. The transmission medium of claim 32 wherein the security key is application and subscriber specific.

44. The transmission medium of claim 32 wherein the transmission medium is a wireless transmission medium.

45. The transmission medium of claim 32 wherein the transmission medium is a wired transmission medium.

46. The transmission medium of claim 45 wherein the wired transmission medium is the Internet.

47. The transmission medium of claim 32 wherein the billing tracking code automatically generates at least one of amount of data sent, amount of data received, a time stamp, an application identifier, a security key, a transaction identifier, and a retry expiration indicator.

48. The transmission medium of claim 32 wherein the billing tracking code is used to route data from the content in accordance with the transmission based billing data.

49. The transmission medium of claim 32 wherein the content is transmitted to a target wireless device.

50. A transmission-based billing system in a computer environment for automatically generating billing data for content that executes on a client device and that transmits data over a network, comprising:

code modifier that instruments the content with billing tracking code, that, when executed on the client device, automatically communicates billing data that reflects the amount of data transmitted over the network.

51. The billing system of claim 50 wherein the billing tracking code tracks the amount of data sent from the instrumented content over the network.

52. The billing system of claim 51 wherein the network is the Internet.

53. The billing system of claim 51 wherein the amount of data is tracked at a logical packet level.

54. The billing system of claim 50 wherein the billing tracking code tracks the amount of data received by the instrumented content from the network.

55. The billing system of claim 54 wherein the network is the Internet.

56. The billing system of claim 54 wherein the amount of data is tracked at a logical packet level.

57. The billing system of claim 50 wherein the content is Java-based.

58. The billing system of claim 50 wherein the content contains byte-code instructions.

59. The billing system of claim 50 wherein the instrumenting is accomplished at a byte-code level of content examination.

60. The billing system of claim 50 wherein the instrumented content includes a security key and wherein the security key is transmitted with the automatically

communicated billing data so that the integrity of the source of the billing data can be verified upon receipt.

61. The billing system of claim 60 wherein the security key is based upon a random number.

62. The billing system of claim 60 wherein the security key is application and subscriber specific.

63. The billing system of claim 60 wherein the security key in instrumented into the content upon receiving a request to download the content.

64. The billing system of claim 50 wherein the computer environment is integrated with a wireless carrier infrastructure.

65. The billing system of claim 50, further comprising causing the instrumented content to be downloaded to a target device over a wireless transmission medium.

66. The billing system of claim 65 wherein the content is requested by a subscriber of a carrier from the computer environment over a wireless transmission medium.

67. The billing system of claim 50, further comprising causing the instrumented content to be downloaded to a target device over a wired transmission medium.

68. The billing system of claim 67 wherein the wired transmission medium is the Internet.

69. The billing system of claim 50 wherein the billing data comprises at least one of amount of data sent, amount of data received, a time stamp, an application identifier, a security key, a transaction identifier, and a retry expiration indicator.

70. The billing system of claim 50 wherein the billing data is automatically communicated on the transmission basis to a billing server system.

71. The billing system of claim 70 further comprising a packet detecting and forwarding module for transmitting data that is not billing data directly between the subscriber device and a server system that is not the billing server system.

72. The billing system of claim 50, further comprising integrating the billing data with customer-based billing information to generate a customer data record.

73. The billing system of claim 50 wherein the billing data is used to support a plurality of billing policies.

74. The billing system of claim 73 wherein the billing policies include a promotional offer that provides reduced charges for a designated application.

75. The billing system of claim 73 wherein the billing policies are provided by a content provider.

76. The billing system of claim 50 wherein the billing data are used to provide royalty payments to providers of the content.

77. The billing system of claim 50, further comprising causing data transmitted between the instrumented content and the network to be routed in accordance with the transmission-based billing data.

78. The billing system of claim 77 wherein the routing enables efficient use of resources on the network.

79. The billing system of claim 77 wherein a priority is assigned to the content based upon transmission usage.

80. The billing system of claim 50 wherein the billing tracking code incorporates a proxy store and forward technique to transmit billing data and data transmission packets between the instrumented content and a plurality of server systems.

81. A computer-readable memory medium containing instructions for controlling a computer processor in a wireless device to automatically transmit packet-based billing data on a per-content basis, by:

when a packet of data is received by content from a network, logging the amount of data received with an identifier of the content;

when a packet of data is to be sent by the content over the network, logging the amount of data to be sent with an identifier of the content; and

transmitting the logged amount of data with the identifier of the content to a server system to be accumulated, thereby enabling the server system to bill a subscriber based upon the accumulated data.

82. The computer-readable memory medium of claim 81 wherein the logging the amount of data is performed by code that is transparently loaded onto the wireless device.

83. The computer-readable memory medium of claim 81 wherein the logging the amount of data and transmitting the logged data are performed by code that resides in a code library.

84. The computer-readable memory medium of claim 81 wherein the logging the amount of data and transmitting the logged data are performed by code that resides in the network driver software of the wireless device.

85. The computer-readable memory medium of claim 81 wherein the logging the amount of data and transmitting the logged data are performed by code that is written to a specification for transmission-based billing.

86. The computer-readable memory medium of claim 81 wherein the logging the amount of data and transmitting the logged data are performed by code that is instrumented into the instructions prior to execution of the instructions on the client device.

87. A method in a wireless device for automatically transmitting packet-based billing data, comprising:

when a packet of data is received from a network, logging the amount of data received;

when a packet of data is to be sent over the network, logging the amount of data to be sent; and

transmitting the logged amount of data to a server system to be accumulated, thereby enabling the server system to bill a subscriber based upon the accumulated data.

88. The method of claim 87 wherein the logging the amount of data is performed by code that is transparently loaded onto the wireless device.

89. The method of claim 87 wherein the logging the amount of data and transmitting the logged data are performed by code that resides in a code library.

90. The method of claim 87 wherein the logging the amount of data and transmitting the logged data are performed by code that resides in the network driver software of the wireless device.

91. The method of claim 87 wherein the logging the amount of data and transmitting the logged data are performed by code that is written to a specification for transmission-based billing.

92. The method of claim 87 wherein the logging the amount of data and transmitting the logged data are performed by code that is instrumented into the instructions prior to execution of the instructions on the client device.

93. A wireless device that automatically transmits packet-based billing data, comprising:

application with billing and tracking code that,

when a packet of data is received from a network, logging the amount of data received;

when a packet of data is to be sent over the network, logging the amount of data to be sent; and

transmitting the logged amount of data to a server system to be accumulated, thereby enabling the server system to bill a subscriber based upon the accumulated data.

94. The wireless device of claim 93 wherein the billing and tracking code is transparently loaded onto the wireless device.

95. The wireless device of claim 93 wherein the billing and tracking code resides in a code library.

96. The wireless device of claim 93 wherein the billing and tracking code resides in the network driver software of the wireless device and is invoked from the application.

97. The wireless device of claim 93 wherein the billing and tracking code is written to a specification for transmission-based billing.

98. The wireless device of claim 93 wherein the billing and tracking code is instrumented into the application prior to execution of the application.

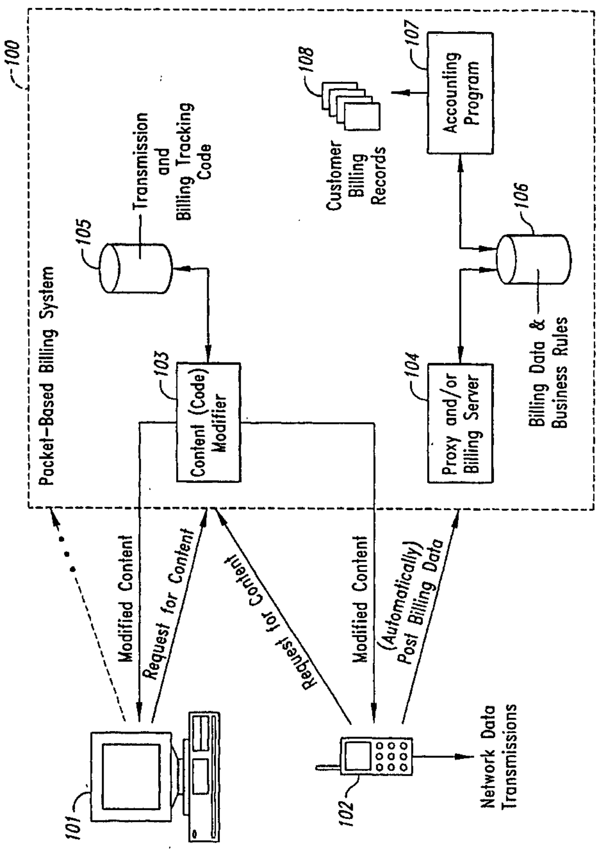

100
Packet-Based Billing System
105
Transmission and Billing Tracking Code
108
Customer Billing Records
107
Accounting Program
106
103
Content (Code) Modifier
104
Proxy and/or Billing Server
Billing Data & Business Rules
Modified Content
Request for Content
Request for Content
Modified Content
(Automatically) Post Billing Data
101
102
Network Data Transmissions

Fig. 1

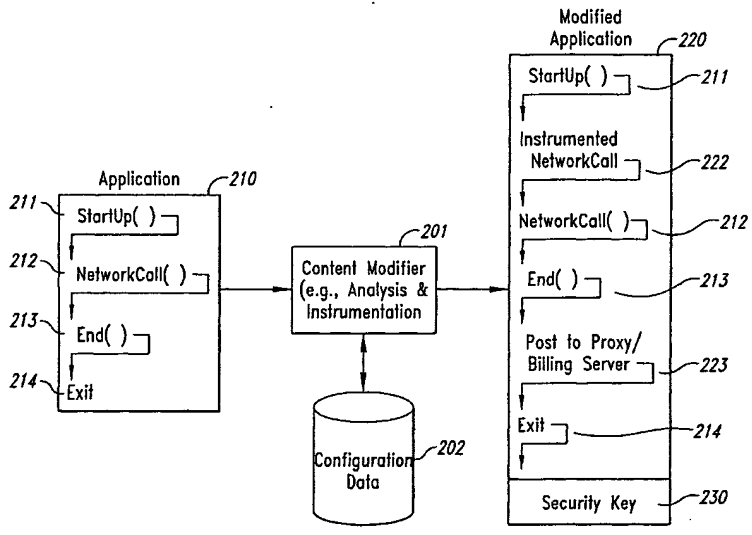

Application
210
211
StartUp( )
212
NetworkCall( )
213
End( )
214
Exit
201
Content Modifier
(e.g., Analysis &
Instrumentation
Configuration
Data
202
Modified
Application
220
StartUp( )
211
Instrumented
NetworkCall
222
NetworkCall( )
212
End( )
213
Post to Proxy/
Billing Server
223
Exit
214
Security Key
230

Fig. 2

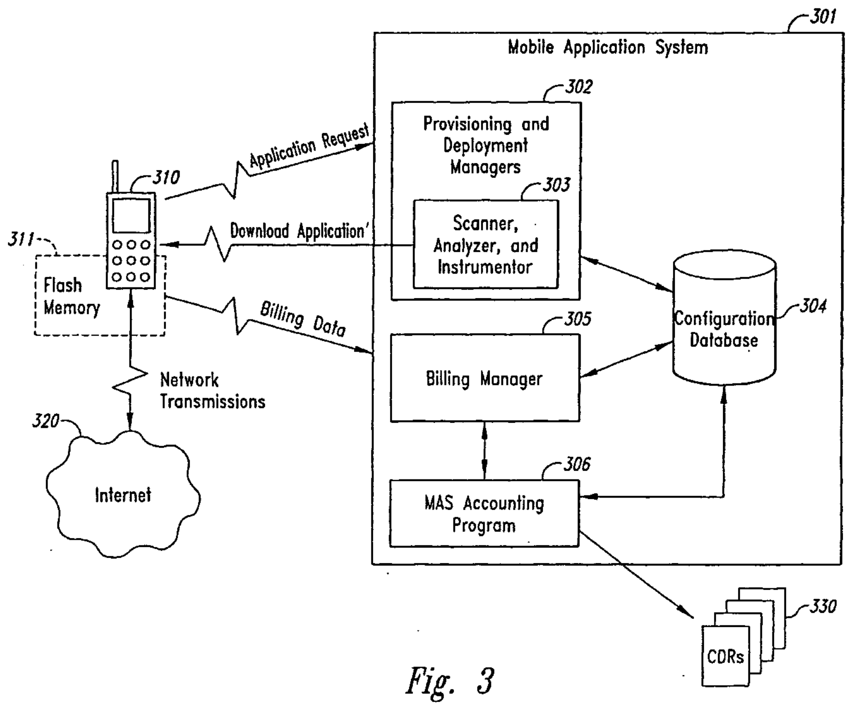

301
Mobile Application System
302
Provisioning and
Deployment
Managers
303
Scanner,
Analyzer, and
Instrumentor
305
Billing Manager
306
MAS Accounting
Program
Configuration
Database
304
330
CDRs
310
311
Flash
Memory
Application Request
Download Application'
Billing Data
Network
Transmissions
320
Internet

Fig. 3

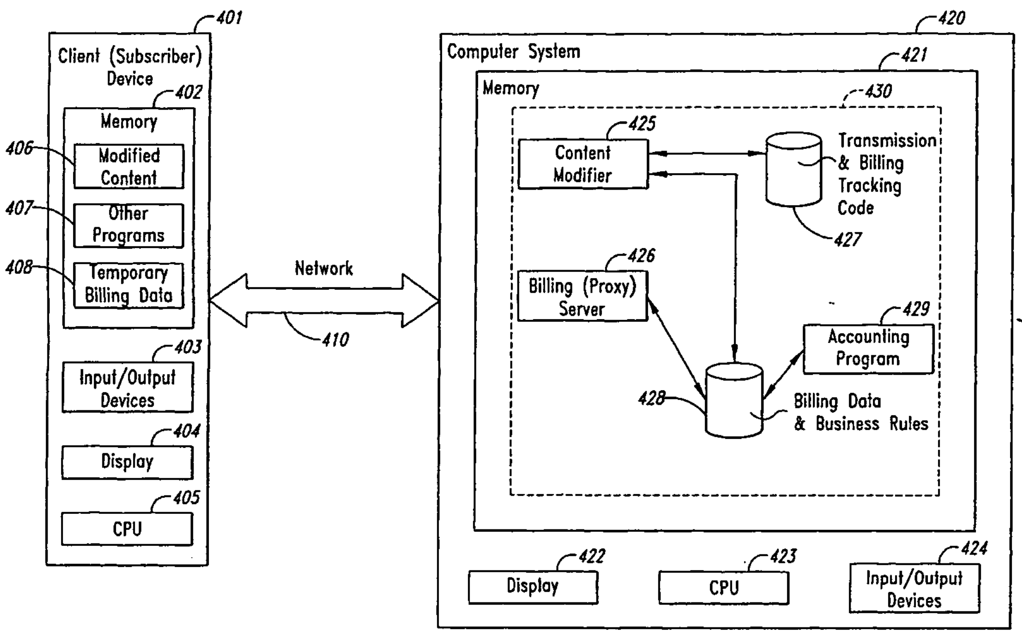

401
Client (Subscriber) Device
402
Memory
406
Modified Content
407
Other Programs
408
Temporary Billing Data
403
Input/Output Devices
404
Display
405
CPU
Network
410
420
Computer System
421
Memory
430
425
Content Modifier
Transmission & Billing Tracking Code
427
426
Billing (Proxy) Server
429
Accounting Program
428
Billing Data & Business Rules
422
Display
423
CPU
424
Input/Output Devices

Fig. 4

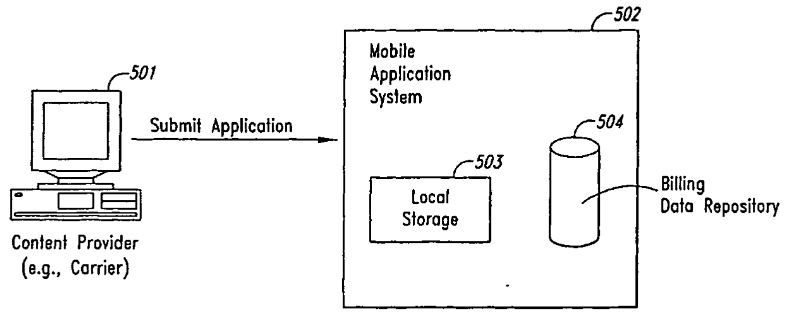

501
Content Provider (e.g., Carrier)
Submit Application
502
Mobile Application System
503
Local Storage
504
Billing Data Repository

Fig. 5

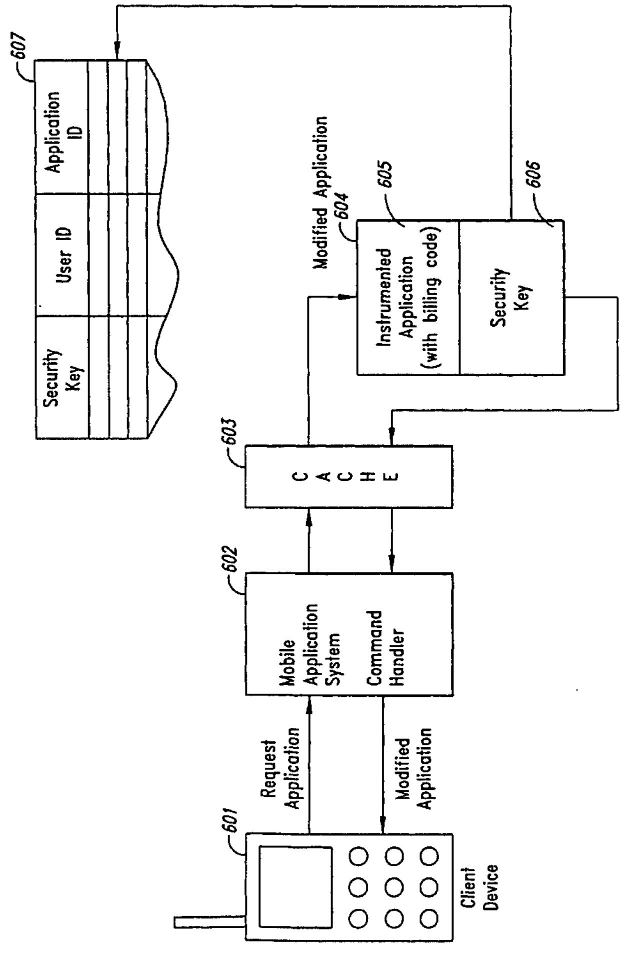

607
Application ID
User ID
Security Key
Modified Application
604
605
Instrumented Application (with billing code)
606
Security Key
603
C A C H E
602
Mobile Application System
Command Handler
Request Application
Modified Application
601
Client Device

Fig. 6

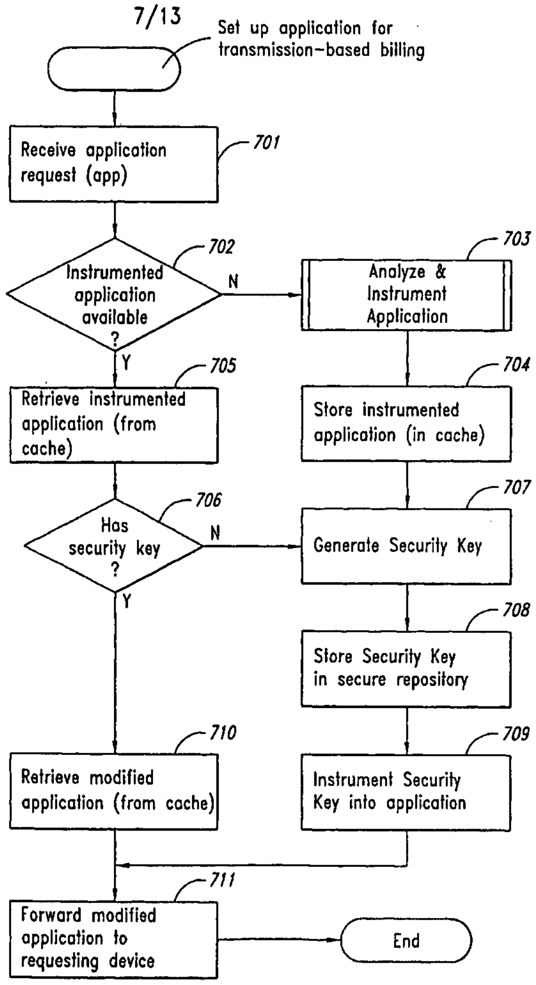

Set up application for transmission-based billing
Receive application request (app)
701
Instrumented application available ?
702
N
Y
Analyze & Instrument Application
703
Retrieve instrumented application (from cache)
705
Store instrumented application (in cache)
704
Has security key ?
706
N
Y
Generate Security Key
707
Store Security Key in secure repository
708
Retrieve modified application (from cache)
710
Instrument Security Key into application
709
Forward modified application to requesting device
711
End

Fig. 7

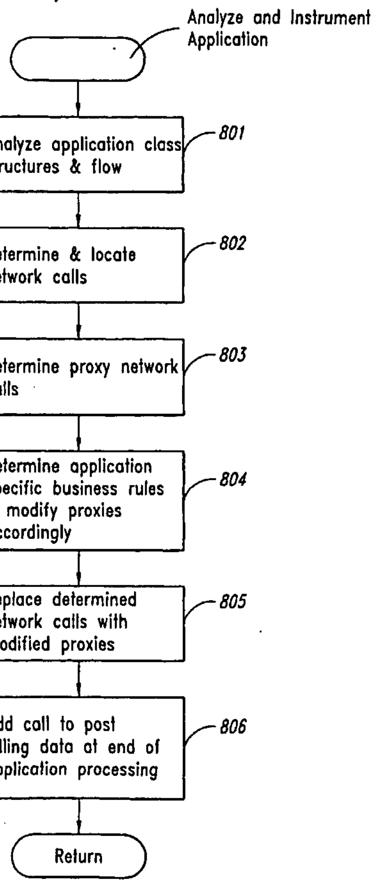

Analyze and Instrument Application
Analyze application class structures & flow
801
Determine & locate network calls
802
Determine proxy network calls
803
Determine application specific business rules & modify proxies accordingly
804
Replace determined network calls with modified proxies
805
Add call to post billing data at end of application processing
806
Return

Fig. 8

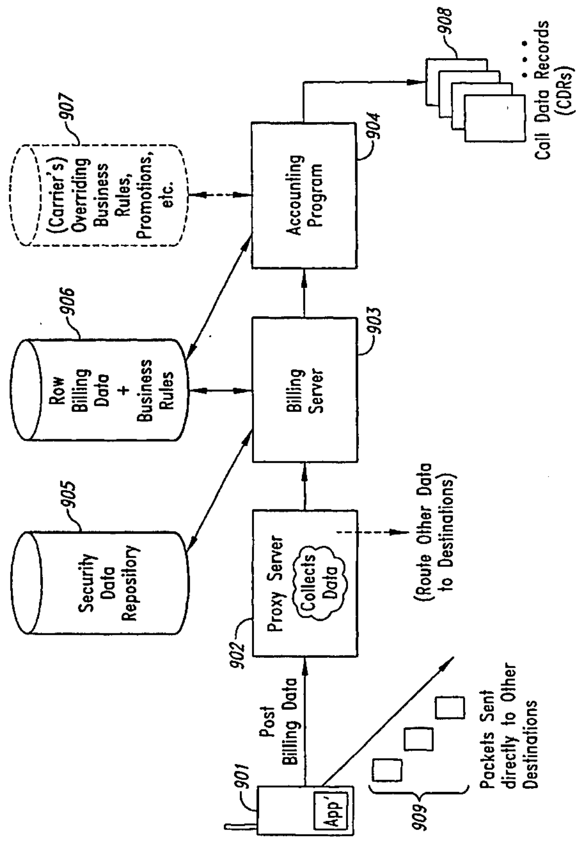

(Carrier's) Overriding Business Rules, Promotions, etc.
907
Row Billing Data + Business Rules
906
Security Data Repository
905
Accounting Program
904
Billing Server
903
Proxy Server
Collects Data
902
(Route Other Data to Destinations)
Post Billing Data
901
App'
909
Packets Sent directly to Other Destinations
908
Call Data Records (CDRs)

Fig. 9

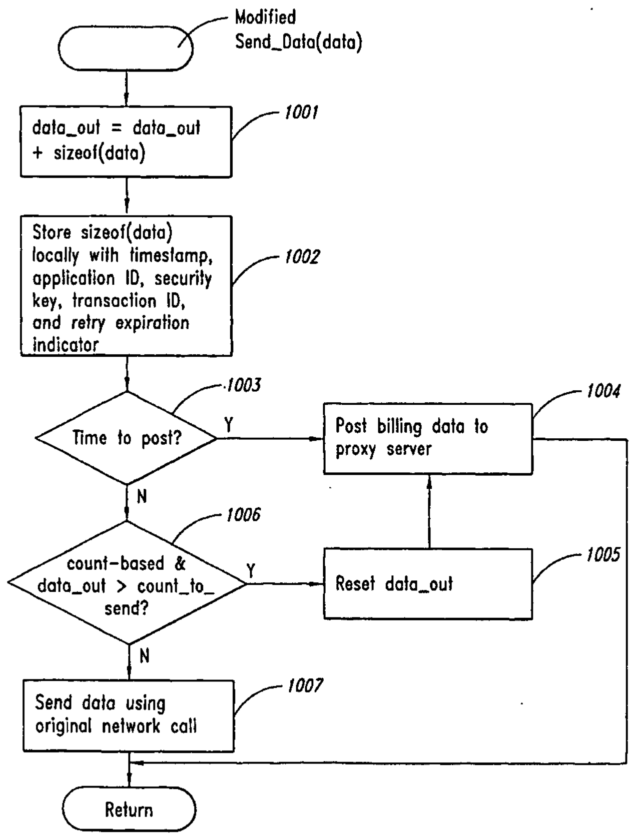

Modified
Send_Data(data)
data_out = data_out + sizeof(data)
1001
Store sizeof(data) locally with timestamp, application ID, security key, transaction ID, and retry expiration indicator
1002
Time to post?
1003
Y
N
Post billing data to proxy server
1004
count-based & data_out > count_to_send?
1006
Y
N
Reset data_out
1005
Send data using original network call
1007
Return

Fig. 10

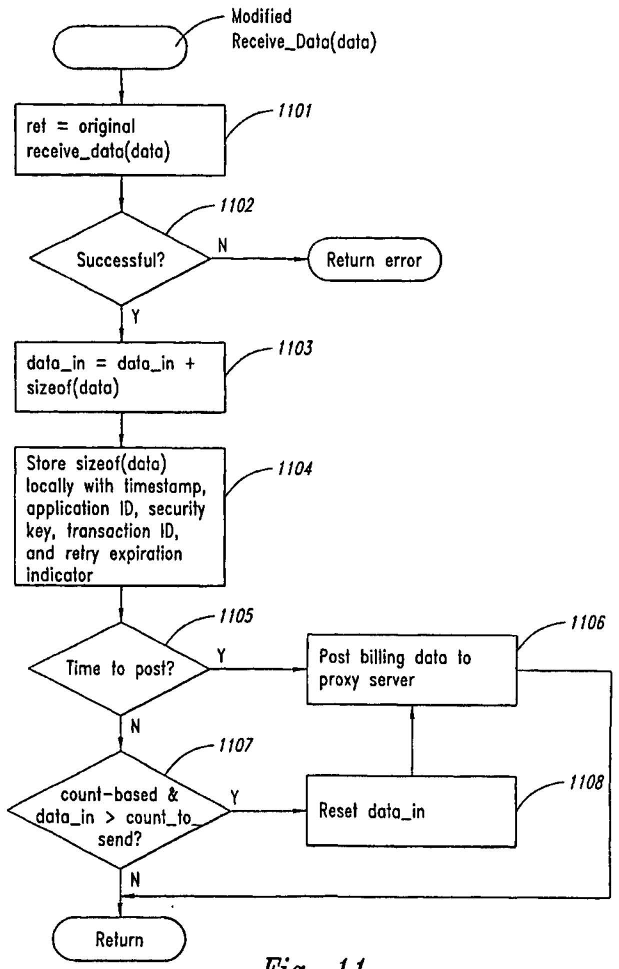
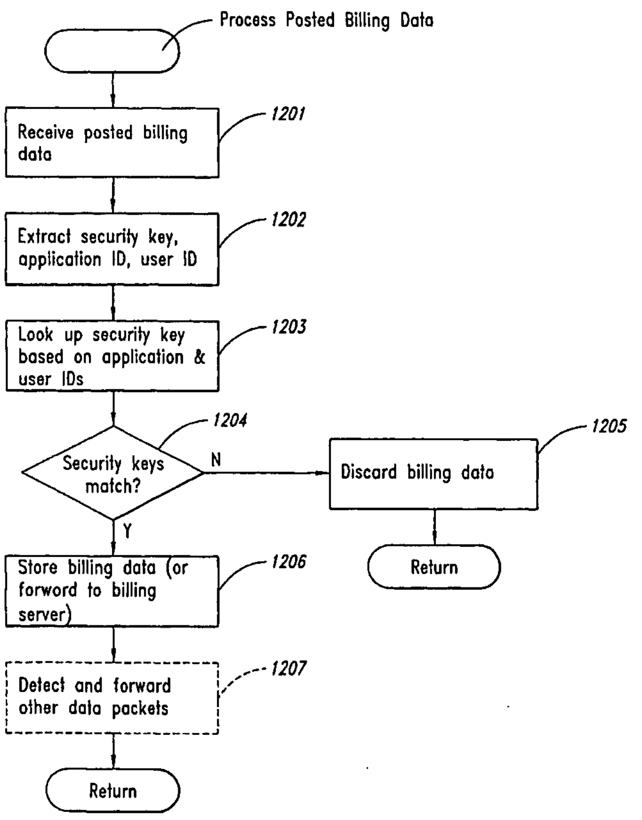
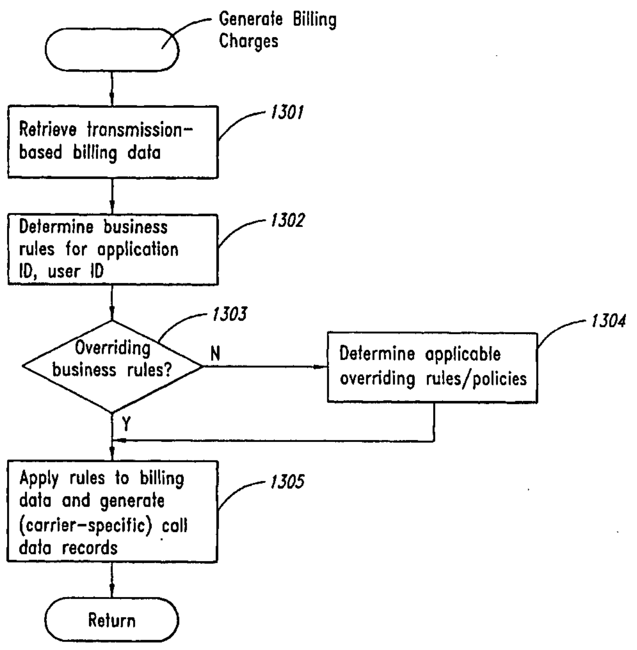

Modified
Receive_Data(data)
ret = original receive_data(data)
1101
Successful?
1102
N
Y
Return error
data_in = data_in + sizeof(data)
1103
Store sizeof(data) locally with timestamp, application ID, security key, transaction ID, and retry expiration indicator
1104
Time to post?
1105
Y
N
Post billing data to proxy server
1106
count-based & data_in > count_to_send?
1107
Y
N
Reset data_in
1108
Return

Fig. 11



Process Posted Billing Data
Receive posted billing data
1201
Extract security key, application ID, user ID
1202
Look up security key based on application & user IDs
1203
Security keys match?
1204
N
Y
Discard billing data
1205
Return
Store billing data (or forward to billing server)
1206
Detect and forward other data packets
1207
Return

*Fig. 12*



Generate Billing Charges
Retrieve transmission-based billing data
1301
Determine business rules for application ID, user ID
1302
Overriding business rules?
1303
N
Y
Determine applicable overriding rules/policies
1304
Apply rules to billing data and generate (carrier-specific) call data records
1305
Return

*Fig. 13*